Higher Performance Neural Networks with Small Floating Point

Learn how to outperform INT8 on Xilinx 16nm and 20nm devices without loss of accuracy or retraining.

ABSTRACT

This white paper introduces a 7-bit small floating point (SFP) number representation that can implement deep neural network models with the same accuracy as INT8 but with 60% higher performance for ResNet-50. This improvement is achieved with an SFP multiplier that can be implemented very efficiently using just 9.5 LUT6s on Xilinx FPGAs. As the DSP blocks are not required for multiplication, the use of SFP results in significantly higher compute density while still achieving a 600MHz clock rate.

The SFP representation is defined, and it will be shown how it can be applied to deep neural networks without requiring any retraining. The best practice for constructing an SFP-based convolution is then described, which makes use of the unique features of the Xilinx DSP block to build efficient large adder trees. Finally, quantitative results compare the area cost and efficiency of different SFP variants.

Introduction

Deep neural networks have brought about significant advances in machine learning, particularly in applications that require object detection, classification, and image segmentation. The quest for ever higher accuracy and image resolutions has led to a constantly expanding zoo of deep learning models, often with higher compute and memory bandwidth requirements.

Specialized hardware architectures are becoming essential to run models within power and latency goals. Xilinx FPGAs offer unparalleled flexibility to develop custom logic and memory architectures, allowing deep neural networks to be tightly coupled to other image processing components. This allows FPGAs to achieve outstanding end-to-end system performance with much lower latency than can be achieved with CPUs or GPUs [Ref 1].

While models are typically developed using 32-bit floating point numbers that follow the IEEE Std 754 standard, it is possible to implement models with 16-bit or even 8-bit numbers without impacting accuracy. Reducing the number of bits required for representing numbers in a model has a significant impact on overall performance as well as power consumption, both in terms of external memory bandwidth and the size of multiply-add logic required(1).

This white paper introduces the small floating point (SFP) number representation, which can implement deep neural network models with 60% more performance than INT8. SFP reduces the size of the multipliers required to the point where it can be implemented efficiently using Xilinx LUT logic. As well as being power efficient, this frees up the DSP blocks so they can be used to perform other functions. The abundance of LUTs in a Xilinx FPGA means that more SFP multipliers can be implemented than would otherwise be possible if DSP blocks were required.

Migrating a deep neural network model from 32-bit floating point to a reduced bit-width representation involves quantization, and quantization can generate discrepancies that might incur a noticeable loss of accuracy. Retraining a model in the context of its new number representation is often used to regain accuracy, however, this is usually a lengthy compute intensive process that can takes days to complete. Table 1 shows that a 7-bit SFP representation is able to achieve the same accuracy as INT8 on the ResNet-50 models without retraining.

Table 1: **Accuracy and Performance Achieved for ResNet-50 [Ref 2]**

	Top1 Accuracy (Normalized)	**Performance on a VU13P**
Float32	100.0%	–
Int8	98.8%	25TOPS
SFP<3,3>	98.7%	40TOPS

1. The size of a multiplier grows quadratically with input bit-width: a 16-bit multiplier is typically four times bigger and more power hungry than an 8-bit multiplier.

Small Floating Point (SFP)

Deep learning models rely on multi-dimensional arrays of real number (tensors). Tensors are commonly implemented with numbers using IEEE Std 754 32-bit floating point representation, however, this white paper shows that high accuracy models can be achieved using short floating point (SFP) representation.

SFP is a floating-point format that is close to 8-bits in size. By using a floating-point format rather than a fixed-point format, such as INT8, a larger range of real numbers can be encoded at the expense of some precision in the larger numbers. For example, a 7-bit SFP can encode a range of numbers that could only be matched by a fixed-point format of at least 13 bits. This white paper shows that increasing the range can have a greater beneficial impact on the deep learning model accuracy than increasing the precision.

Figure 1 shows how a 7-bit SFP<3,3> compares to the 32-bit and 16-bit IEEE Std 754 floating point formats. Floating point numbers contain a sign (s), an exponent (e) and a mantissa (m). The mantissa is an unsigned fixed-point value that has been shifted such that the most significant bit is always '1'; this leading bit is often omitted as it is implicit. The unsigned exponent indicates the position of the binary point in relation to the mantissa's leading bit, thus extending the range of numbers that can be represented. It is common for the exponent to be 'biased' to allow the binary point to moved left as well as right.

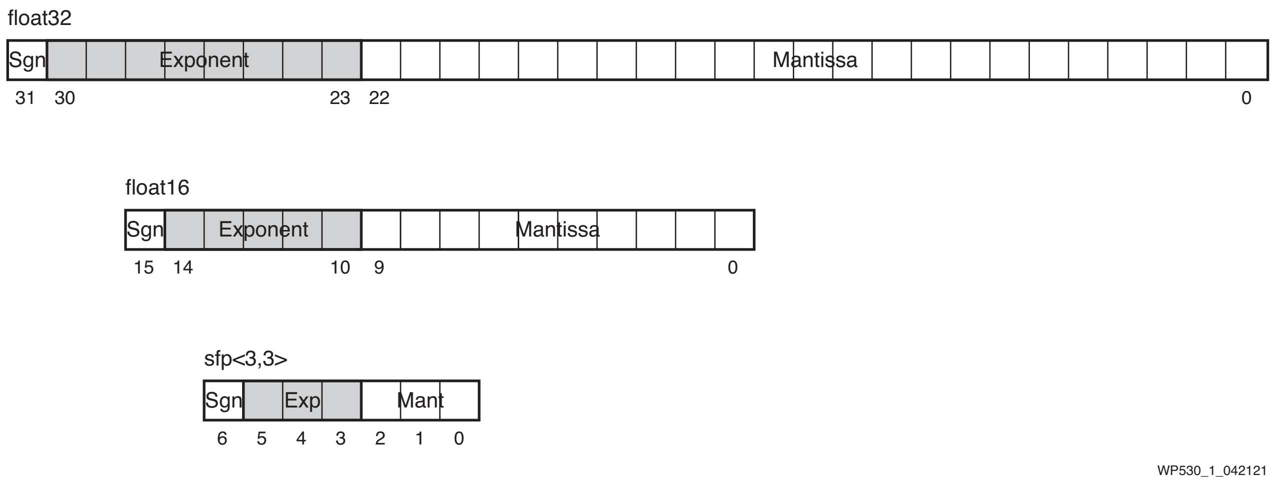

Figure 1: **Bit Layout of Different Floating Point Number Representations**

Equation 1 shows the formula relating a number in a floating-point format with the real value, r, it represents.

$$r = (-1)^{sign} * 2^{(exp-BIAS)} * \left(1 + \frac{\Sigma(mant_i * 2^i)}{1 + \Sigma 2^i}\right)$$

Equation 1

The 7-bit SFP<3,3> format uses 3-bits for the exponent and 3-bits for the mantissa; a bias value of 4 is used, and an exponent value of zero is reserved to represent the real value of 0.0. Hence, the smallest non-zero value that can be represented in SFP<3,3> is ±0.125, and the maximum values are ±15.

Zero and SubNormals: In IEEE Std 754 float32, an exponent value of zero is reserved for 0.0 and subnormal numbers. Subnormal numbers have an implicit leading mantissa bit of '0' rather than '1,' extending the range of the smallest numbers but at the expense of further precision. Subnormal numbers have a significant impact on the hardware required to implement multiplication and addition, and did not impact the deep learning model's accuracy in experiments, therefore, the SFP format does not use subnormals, and treats all numbers with an exponent value of zero as 0.0.

Infinity and Not-A-Number (NaN): In float32, IEEE Std 754, the maximum exponent value (255) is reserved to represent Infinity or NaN. In these cases, the mantissa is also used for additional information such as whether the number generates run-time exceptions when used ('signaling' or 'quiet'). These special values are not used in SFP, so the maximum exponent value is not reserved. All SFP maths that would overflow should saturate to the highest or lowest value.

Exponent Bias: The usual convention is to use a bias of (2^{e-1} -1), which centers the value 1.0 as close to the median value so that there are as many different numbers below 1.0 as there are above it. This convention is useful for multiplication where the output floating-point format is the same as the input floating-point format as 1.0 remains the natural median value. However, SFP multiplication will extend the exponent bits to avoid exponent overflow, and it is simpler and more hardware efficient to use a bias of (2^{e-1}).(1)

Motivation for SFP

Figure 2 shows the distribution of fixed weight values used in ResNet-50 for an 8-bit fixed point representation such as INT8. The distribution is far from even with over 99% of the weights occurring in the smallest 20% of values. Although the larger values are much less common, they are as essential as the small values to achieve high model accuracy.

Interestingly, the logarithmic distribution is almost linear (shown by the line in Figure 2), indicating that the occurrence of each INT8 value is inversely related to its magnitude. This is the kind of relationship that floating point formats encode very efficiently because they devote more coded values to smaller numbers, while larger numbers are represented with lower precision.

1. With a bias of (2^{e-1} -1), SFP<3,3> has a maximum value of 30, but SFP<4,3> has a maximum value of 480, which is not high enough to represent 30x30. However, with a bias of (2^{e-1} -1), SFP<3,3> has a maximum value of 15, and SFP<4,3> has a maximum value of 240, which is high enough to represent 15x15.

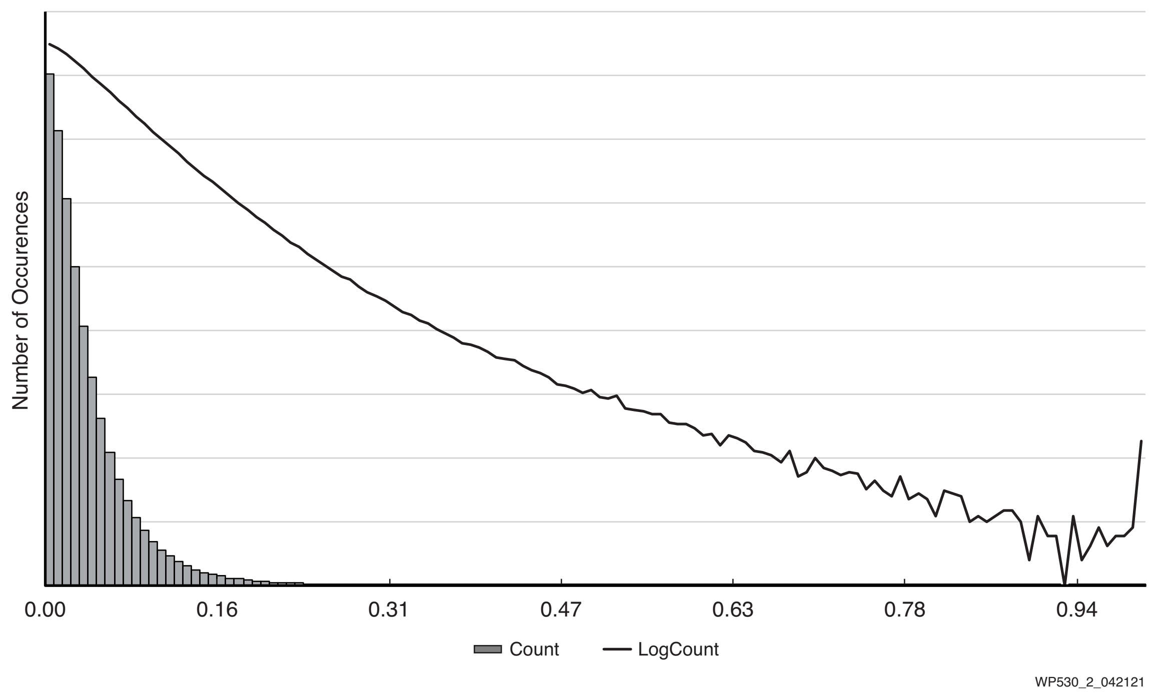

Figure 2: **Histogram of ResNet-50 Weights (Magnitude) for INT8**

Figure 3 shows the distribution of weights when the SFP<3,3> representation is used. The distribution is far more even than it was for INT8. It is thought that this 'fairer' encoding of model values is the reason why high accuracy can be achieved with an SFP representation even though it uses fewer bits than INT8.

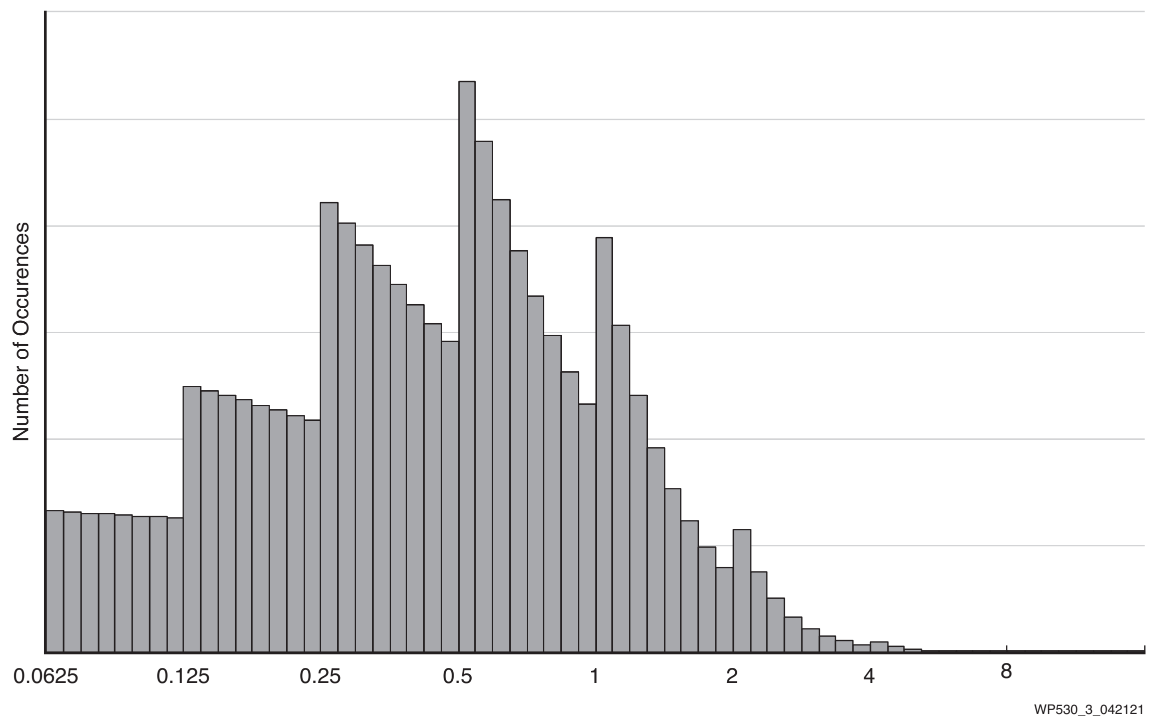

Figure 3: **Histogram of Non-Zero ResNet-50 Weights (Magnitude) in SFP<3,3> Format**

Experiments showed that 2-bit exponent based SFP formats provide insufficient range to work for deep learning models without retraining, however 3-bit exponents are, in fact, slightly over-ranged. This means that between four and eight different exponent values are needed to maintain accuracy. SFP requires one value to be reserved to represent 0.0, but to reduce the amount of hardware needed, all values with a zero exponent (i.e., 0.0625 to 0.125) are used to represent 0.0. Experiments showed that the loss of this range of values (rather than a single value) did not impact overall accuracy.

Efficient SFP Multiplication on Xilinx FPGAs

As well as maintaining a high level of accuracy in ResNet-50, SFP<3,3> is particularly well suited to implementation on Xilinx FPGAs, yielding more than double the density achieved with INT8 in practice. This is possible because the SFP format only requires small multipliers that can be efficiently implemented in Xilinx look-up tables (LUTs) without needing to use specialized DSP multiplier resources.

Figure 4 shows the logic needed to implement SFP<3,3> multiplication. The two numbers being multiplied (A and B) have 3 mantissa bits each, and even when the leading hidden 1 bit is appended, the multiplier logic is in a function of just six different wires. Functions of six wires, no matter how complex, can be implemented very efficiently in Xilinx LUT6s. Once multiplied, the 8-bit result must be normalized and then truncated and rounded to the desired output mantissa width. For SFP<3,3>, the normalization and rounding stages that are usually costly in logic resources can be collapsed into the LUT6s that are already used to implement the multiplier logic, resulting in just a single LUT6 per output bit. This also results in a very shallow logic depth, allowing the multiplier to run at over 600MHz.

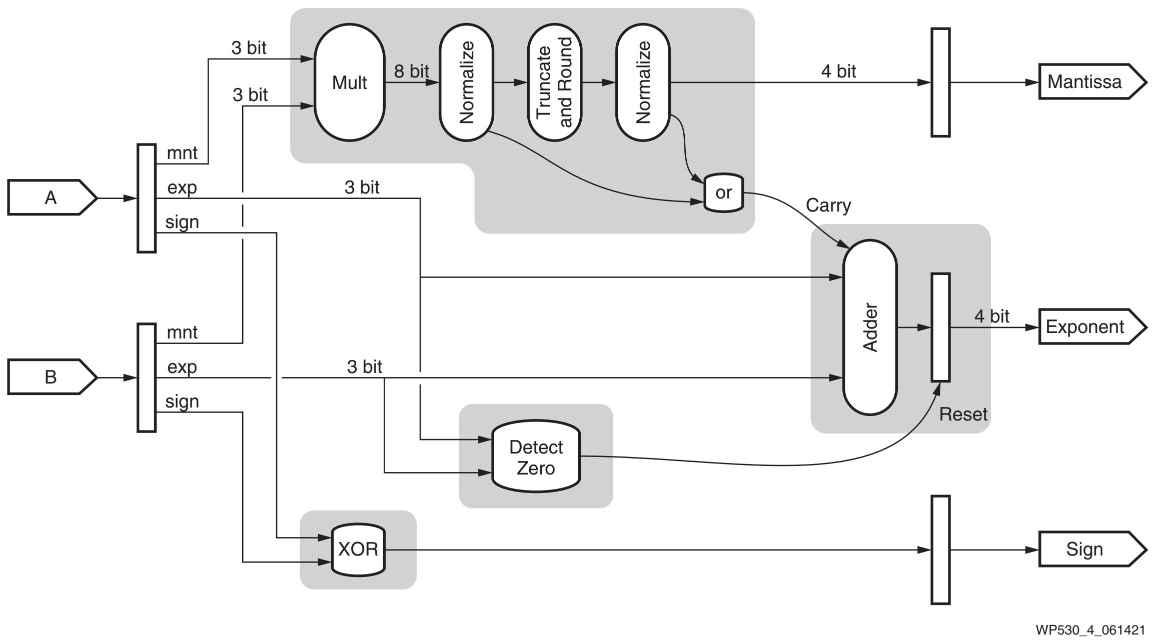

Figure 4: **Implementation of an SFP<3,3> Multiplier**

Normalization adjusts a mantissa result such that there is always a '1' in the leading bit position. Normalization can require halving the mantissa and adding one to the exponent to compensate. Although there are two normalization stages, at most, only one exponent adjustment is ever needed for SFP<3,3>. A single LUT6 can, therefore, be used to determine whether the exponent result should be increased by one as the result of normalization.

Handling 0.0 requires extra logic to detect a zero exponent in either A or B inputs. With SFP<3,3>, this results in a single LUT6, which is used to force the output of the adder to reset to 0.0. Computing the sign bit and the exponent are done with standard Xilinx logic and adder circuitry. The overall implementation requires the equivalent of just 9.5 LUT6s (the XOR can be packed with other logic), which is significantly smaller than even a 4-bit LUT based multiplier, which requires 18 LUTs.

Xilinx white paper, *Deep Learning with INT8 Optimization on Xilinx Devices* [Ref 3] used DSP blocks to implement INT8 multiplication (up to two INT8 multipliers per DSP-block), however, the overall performance was limited by the number of DSP blocks available. With SFP<3,3>, just 19 LUT6s are required to achieve the same performance that required an entire DSP block in INT8. In a Xilinx® Virtex® UltraScale+™ VU13P FPGA, there are 140 LUT6s for every DSP, so the abundance of logic resources allows for significantly more compute density for SFP multiplication.

SFP Quantization without Retraining

Figure 5 shows an example of a deep neural network model used for image classification. An image classification network is a feedforward network that transforms activation tensors in a series of layers. The initial activation tensor is often derived from an image, and the output is a tensor with an entry for each class containing the probability of the object in the image belonging to that class. Convolution layers are typically the most compute intensive layers, and these must be adapted to work with SFP based tensors.

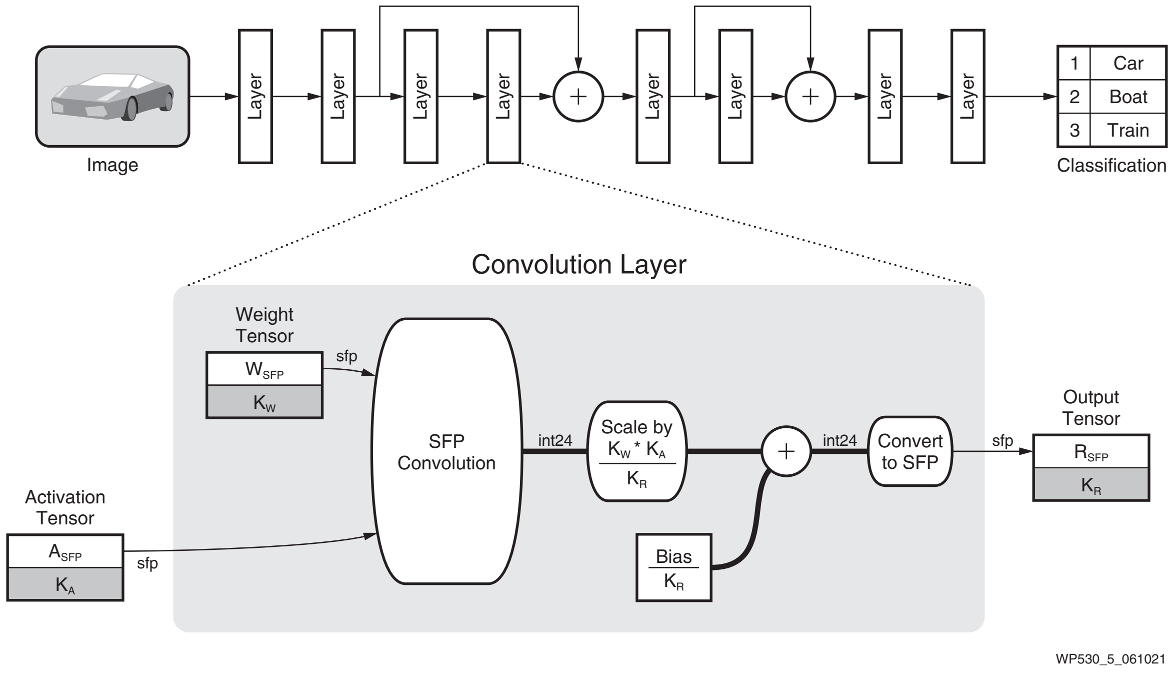

Figure 5: **Example of an SFP Convolution Layer within a Deep Neural Network Model**

Each of the original floating-point tensors is replaced with an SFP-tensor and a scaling factor (K). Different tensors can have different scaling factors, but all SFP values in a particular tensor are scaled by the same factor.(1)

SFP convolution is performed on just the SFP values, and internally, accumulation uses a 24-bit fixed point format. The final 24-bit result must be scaled by the input and output scaling factors before being converted back to an SFP value in order to match the original floating-point behavior. The SFP scaling factor (K_W * K_A / K_R) is represented as a 16-bit fixed point number so that scaling can be done using a single DSP block.

Before mapping a model onto hardware, the K scaling factors must be determined. This is done in a calibration phase where a set of images is presented to the model, and the maximum absolute values observed in each tensor are recorded. INT8 requires that KL-divergence is used to establish calibration values in order to achieve high accuracy [Ref 4], however using the maximum absolute value works very well with SFP. The K values are then chosen to scale these maximum values to the maximum SFP value allowed. This process is applied to weights as well as activations, although the weights are not dependent on the input images and can be processed in advance. Unlike retraining, the images only need to be presented once and the weight values are not updated, so this 'observation' process is far less compute intensive.

Batch normalization layers are often used during training to scale tensors after convolution. To get the best quantization results, batch normalization layers are merged with their preceding convolution layers before the K scaling factors are chosen. This is achieved by adjusting the convolution weights.

Many deep neural network models contain additive layers that sum tensors (of identical shapes) together. To implement additive layers using SFP, the scaling factors must be considered. An economical way of implementing this is to adjust the scaling factors to be identical for all tensors that lead into and out of the additive layer. The additive layer can then be implemented using just SFP-addition without requiring any pre- or post-multipliers. Note that SFP-addition is a floating point addition requiring shifting and renormalization as well as integer addition.

SFP Convolution Implementation

Figure 6 shows the datapath used to perform SFP convolution. All convolutions were processed in hardware in fixed 16x1 tensors. Tensors in deep learning models vary in size and shape so they must be decomposed to match the hardware size, padding with extra zeros if necessary. The memory architecture and sequencing logic to achieve this decomposition is very similar to that used in INT8 and so it is not shown here.

1. In general, the scaling factors (K) are vectors with a separate value for each tensor axis, however, SFP is able to achieve high accuracy with a single per-tensor scaling factor. Both activations and weights each use their own per-tensor scaling factor.

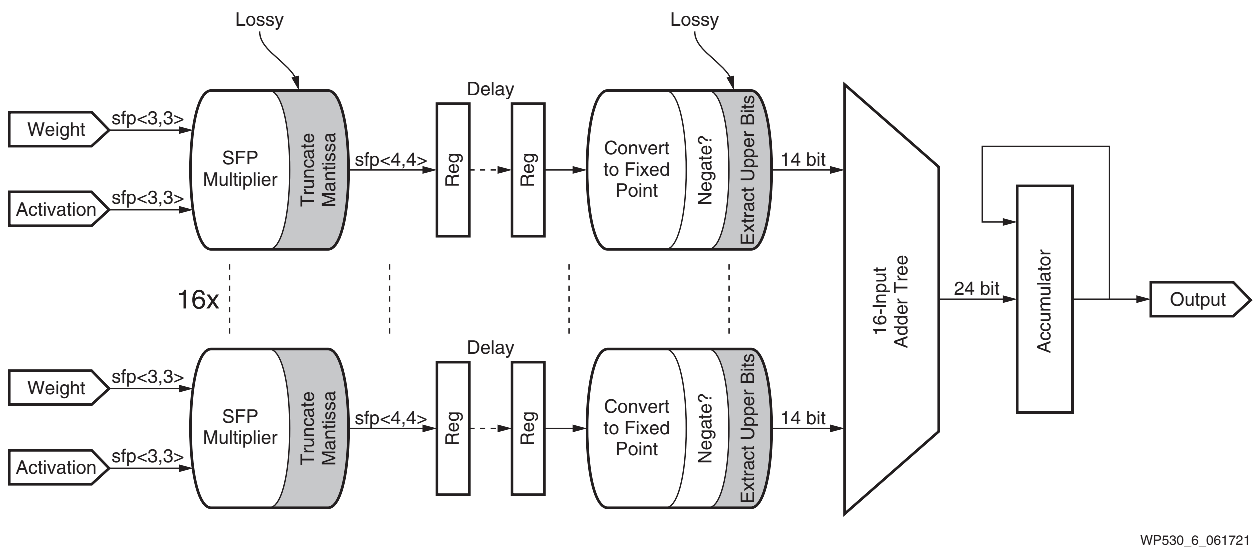

Figure 6: **SFP Convolution Core**

The first stage of the convolution core multiplies the SFP weights and activations together using the circuit shown in Figure 4. The multiplier results are each converted to fixed point before being summed in an adder-tree. Delay registers are used to balance the pipeline depth of each input through the adder tree. The adder tree result often needs to be accumulated over several clock cycles because filters are often much more than 16 taps deep. The convolution core is fully pipelined so that data for a new filter can be clocked in on the next clock after the last data of the previous filter, and the SFP convolution logic is doing useful work on every clock cycle. See Figure 7.

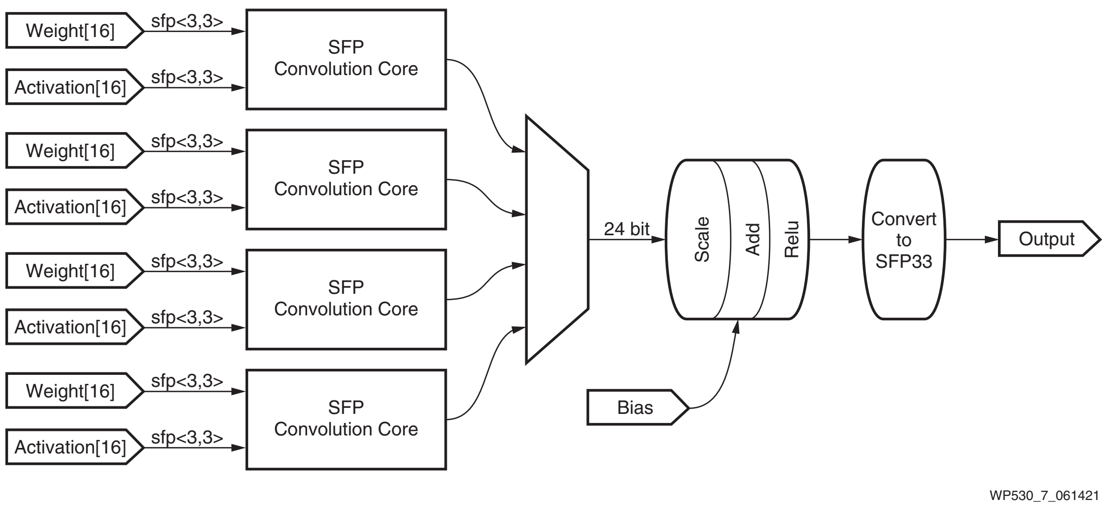

Figure 7: **Post SFP Convolution Scaling and ReLU**

When the last cycle of a filter has been accumulated, this accumulator value is scaled and biased (to compensate for the SFP scaling factors), and then a non-linear ReLU function is applied. The fixed-point result is then converted back into an SFP<3,3> value. Most deep neural network models require several clock cycles to accumulate each filter result, so the post-convolution logic can be shared between several convolution cores, reducing the overall logic required. For the results in this white paper, eight convolution cores share the same post-processing logic.

Very high model accuracy is achieved, even though the convolution core involves several lossy processes (shown in gray in Figure 6). Extensive experimentation was done to establish the best parameters to use to maintain accuracy while also reducing the number of LUTs needed:

- The SFP multiplier preserves exponent accuracy by using a 4-bit exponent output and a bias value of 4 (rather than a more conventional 3). However, the SFP multiplier truncates the output to 4 bits (from 7 bits), reducing the size of the logic needed to convert the SFP output to fixed point.
- The pipeline balancing delay registers are inserted between the SFP multiplier and the fixed-point converters because this is where the values are in their most compact representation.
- The fixed-point conversion requires shifting the mantissa by the 4-bit exponent and negating the result based on the sign bit. Negation is approximated by just inverting the bits, which requires less logic than would be needed implementing an adder-based negation. Only the top 14 bits of the 21-bit fixed point value are necessary to maintain accuracy.

The convolution core can be implemented very efficiently using just conventional logic. Previous deep neural network architectures have been limited by the number of DSP blocks on a device, but the compact SFP multiplier is not constrained by this resource and is, hence, able to achieve much higher compute density. The now unused DSP blocks can be re-purposed to implement the adder-tree and accumulator in the SFP convolution core. This balances the resource usage between LUTs and DSPs to deliver the ultimate in compute density on Xilinx devices.

Using Xilinx DSP Blocks to Maximize SFP Performance

The DSP block in Xilinx devices is one of the most versatile multiplier blocks available in FPGAs. While most use-cases for DSP are for multiplier-based structures, it can also be configured to implement wide logic functions as well as pipelined adders. For applications where there is not such a great need for large multipliers, the Xilinx DSP block can be an effective way to improve overall logic density and alleviate LUT usage and congestion.

Figure 8 shows a DSP block that has been configured to perform two independent 24-bit additions (using the TWO24 SIMD mode). The DSP block can also be configured to make use of built-in registers to improve the speed and reduce the number of registers required in the programmable logic. The *hard* multiplier of the DSP block is unused in this configuration and draws very little power.(1) The A and B inputs to the DSP block that usually feed the multiplier instead are treated as 2x24-bit input that feeds the X-multiplexer of the dual 24-bit adder.

1. Set parameter USE_MULT to NONE when using only the adder/logic unit to save power [Ref 5].

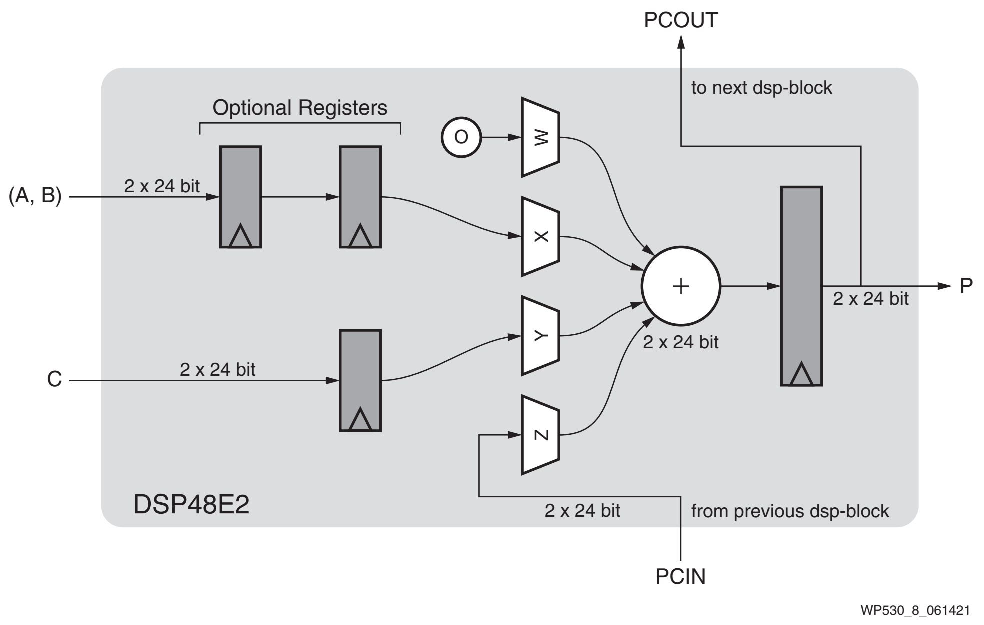

Figure 8: **Xilinx DSP Configured as a Dual 24-Bit Pipelined Adder**

Xilinx DSP blocks are laid out in columns on the silicon die, and dedicated column wiring can be used to cascade DSP blocks together to create larger structures. These cascade connections are an efficient way of creating larger adder trees used in the SFP convolution core. Figure 9 shows how three DSP blocks can be cascaded to build a 6:1 adder tree (in fact, this is a pair of 6:1 24-bit adder trees as it is used in the SIMD mode). Embedded DSP registers are used as much as possible to achieve maximum clock frequency while reducing the number of external pipeline stages required in the programmable logic. This structure of just three DSP blocks replaces the equivalent of 240 LUT6s.

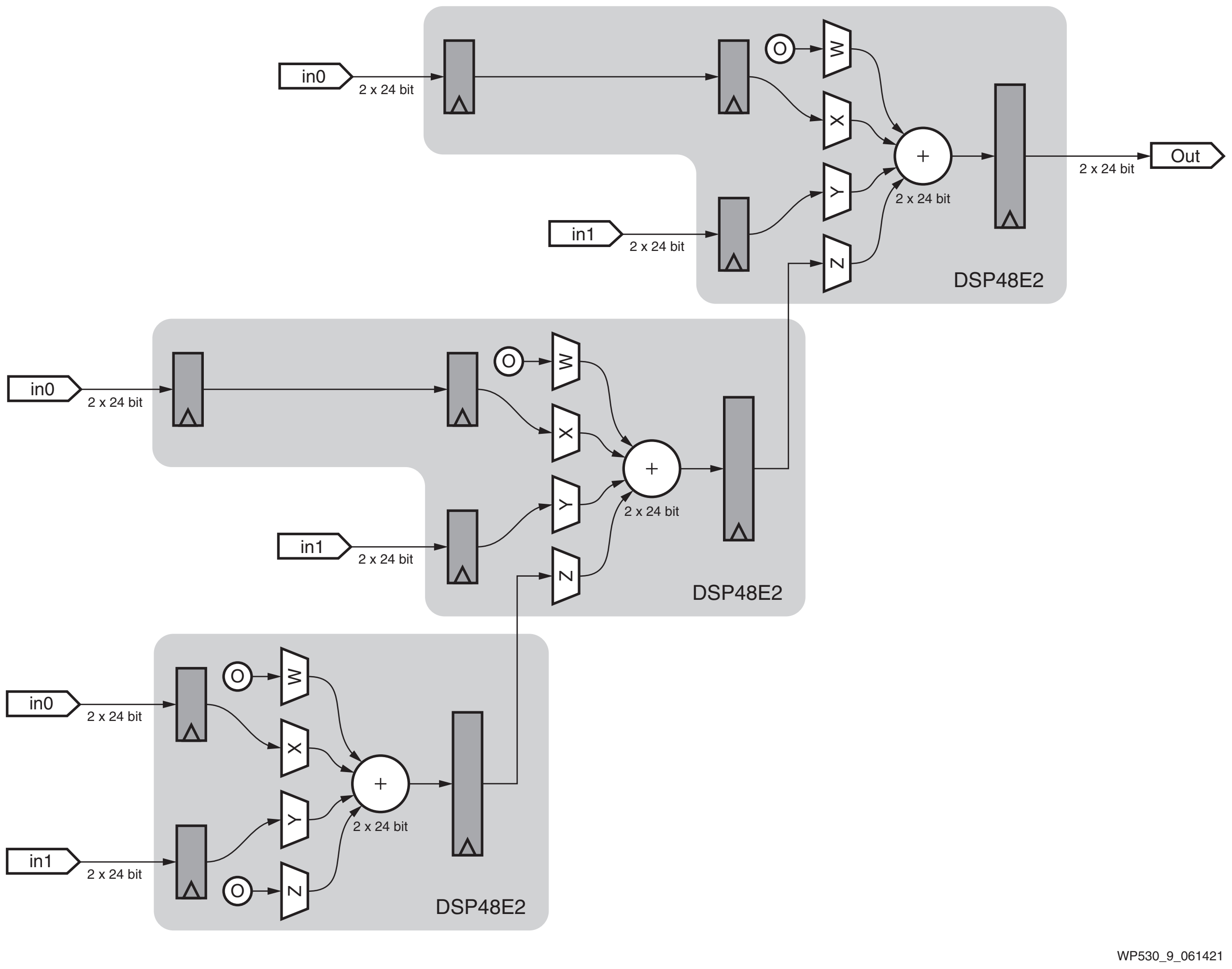

Figure 9: **Cascaded DSP Blocks Form a Pair of 6:1 24-Bit Pipelined Adders**

Although larger cascade chains of DSP blocks are possible, the linear structure means that the pipeline depth increase requires more and more registers in the programmable logic. A tree structure becomes more efficient to use even though some connections must then be routed using programmable logic routing resources rather than the dedicated DSP cascade chain.

Figure 10 shows a complete dual 16:1 adder tree (including the accumulators) used for two SFP convolution cores. Each DSP block is implemented via two independent 24-bit datapaths: one datapath for all the white inputs and output; the other datapath for all the dark gray inputs and output. It is not possible to balance the pipeline depth to all the inputs, so the SFP multipliers must be constructed with additional pipeline registers dependent on their location (this is achieved with the extra registers that were shown in Figure 6). The final DSP is configured to feedback its output to the W-multiplexer of its adder to implement an accumulator; the DSP block contains additional control inputs that can force the W-multiplexer to zero for when the accumulator must start processing a new batch of values.

Although the SFP convolution core engine makes use of DSP blocks, it only requires 10 DSP blocks to implement 2 x 16 tensor values per clock. This compares to INT8 where at least 16 DSP blocks (in a long cascade chain) would be required, hence, this implementation yields a 60% density advantage.

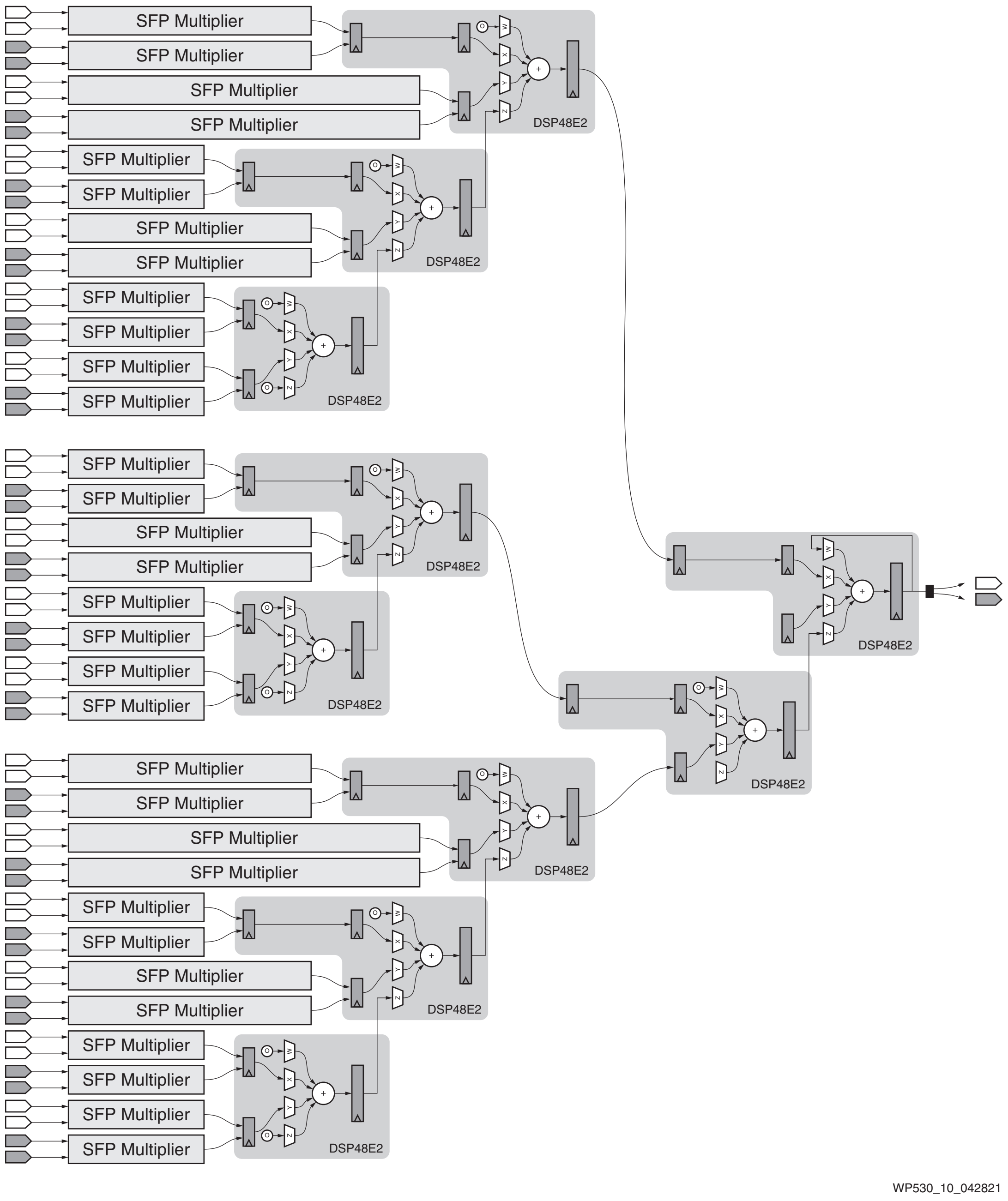

Figure 10: **Implementing a Dual 16:1 SFP Convolution Core Using DSP Blocks**

Quantitative Analysis of SFP

TensorFlow 2.3.1 was used to conduct experiments on different SFP representations, and different lossy truncation behavior. As SFP is not native to CPU or GPU, custom layers were created in C++ to emulate the SFP calculations needed. In all experiments, all layers used the same SFP representation, and no retraining was performed. The scaling factors (K) were calibrated based on only the first 1,000 examples.

Table 2 shows the accuracy achieved for different SFP representations. In each case, the SFP multiplier output has one more exponent bit and one more mantissa bit than the base SFP representation, and no fixed-point truncation was performed. The table shows that at least three exponent bits and two mantissa bits are needed to achieve reasonable model accuracy, but three mantissa bits is sufficient to achieve full accuracy. Perhaps surprisingly, increasing the number of exponent bits and decreasing the number of mantissa bits does not seem have a strong effect on accuracy.

Table 2: **Accuracy Achieved when Quantizing Both Activations and Weights to SFP<e, m>**

Top1 Accuracy (Normalized to FP32)		**Number of Mantissa Bits (m)**					
		0	**1**	**2**	**3**	**4**	**5**
Number of Exponent Bits (e)	2	0%	0%	0%	0%	0%	0%
	3	0%	65.2%	94.5%	98.8%	100.4	100.5%
	4	0%	53.3%	95.9%	99.1%	100.6	–
	5	0%	54.8%	95.7%	99.2%	–	–
	6	0%	53.6%	95.7%	–	–	–

Table 3 shows several models with different formats and design parameters ranked based on accuracy achieved. The number of LUTs required to implement the dual 16-tap convolution core shown in Figure 10 is also shown (this was obtained directly from the Vivado® Design Suite's post-placement).

Table 3: **Comparison of Area and Accuracy for Different Design Parameters**

Format	**Format Bits**	**Multiplier Mantissa Bits**	**LUTs**	**DSP Blocks**	**Top1 Accuracy (Normalized to FP32)**
INT8	8	–	–	16	98.5%
SFP<3,3>	7	4	872	10	98.1%
SFP<4,3>	8	4	980	10	98.4%
SFP<4,2>	7	3	877	10	95.9%
SFP<3,2>	6	3	726	10	94.5%
SFP<3,3>	7	3	770	10	96.6%

Table 3 shows that the SFP<3,3> format used in this white paper has excellent accuracy at a reasonable cost and uses 38% fewer DSP blocks than INT8. Its 7-bit number format can also be exploited to reduce memory traffic by 12.5% compared to INT8 and should lead to a similar reduction in power usage. Note that for the results in Table 3, the multiplier output was truncated to a 14-bit fixed-point number. This truncation yields a small loss in accuracy when compared to Table 2 where no truncation was performed (0.7% for SFP<3,3>).

As an alternative, the SFP<3,2> format might be a good choice for applications that can afford to sacrifice 3.5% accuracy to achieve significant memory, power (–25%) and LUT (–16.7%) savings.

Conclusion

This white paper introduced a 7-bit SFP<3,3> number representation that can implement deep neural network models with the same accuracy as INT8 for ResNet-50 but with 60% higher performance. This improvement is achieved with an SFP multiplier that can be implemented very efficiently using just Xilinx FPGA LUTs. The DSP blocks that would normally be used for multiplication are re-purposed to build efficient large adder trees used in SFP convolutions.

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

Acknowledgments

The following have authored or contributed to this white paper:

Dr. Paul Metzgen, Silicon Tailor, Founder and CEO

Shaoxia Fang, Xilinx Director, DPU Microarchitecture

Satyaprakash Pareek, Xilinx Senior Software Engineer

Bing Tian, Xilinx Principal Engineer

Yi Shan, Xilinx Senior Director, AI

Elliott Delaye, Xilinx Distinguished Engineer

Ashish Sirasao, Research Scientist, AI-Codesign team at Facebook

References

1. Xilinx GitHub, ML Performance Inference_Results_v0.7, https://github.com/mlperf/inference_results_v0.7/tree/master/open/DellEMC.
2. Xilinx GitHub, https://github.com/Xilinx/Vitis-AI/tree/master/models/AI-Model-Zoo/model-list/tf2_resnet50_imagenet_224_224_7.76G_1.3
3. Xilinx White Paper, WP486, "Deep Learning with INT8 Optimization on Xilinx Devices White Paper," https://www.xilinx.com/support/documentation/white_papers/wp486-deep-learning-int8.pdf.
4. Migacz, Szymon, 8-Bit Inference with TensorRT, presentation, GPU Tech Conference, 2017, https://on-demand.gputechconf.com/gtc/2017/presentation/s7310-8-bit-inference-with-tensorrt.pdf.
5. Xilinx User Guide, UG579, "UltraScale™ Architecture DSP Slice," https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf.

Revision History

The following table shows the revision history for this document:

Date	Version	Description of Revisions
06/17/2021	1.0.1	Typographical edit.
06/14/2021	1.0	Initial Xilinx release.

Disclaimer

Automotive Applications Disclaimer