XAPP706 (v1.0) March 31, 2005

Alpha Blending Two Data Streams Using a DSP48 DDR Technique

Author: Reed P. Tidwell

Summary

The full throughput of a Virtex™-4 DSP48 slice can be achieved by time-multiplexing two data streams with a double data rate (DDR) technique. Alpha blending is an example of this technique. This application note describes a reference design file (xapp706.zip) available on the Xilinx web site.

Introduction

The Virtex-4 XtremeDSP™ system feature, embodied as the DSP48 slice primitive, is a high-performance computing element operating at an industry-leading 500 MHz. The design of the Virtex-4 infrastructure supports this rate, with Xesium™ clock technology, Smart RAM, and LUTs configured as shift registers. Many applications, however, do not have data rates of 500 MHz. This application note describes how to harness the full computing performance of the DSP48 slice with data streams of lower rates by using a DDR technique through the DSP48 slice. The DSP48 slice, operating at 500 MHz, can multiplex between two data streams, each operating at 250 MHz. Alpha blending of video data is one application of this technique. Alpha blending combines two streams of video data according to a weighting factor called *alpha*. This article explains techniques and design considerations for applying a DDR technique to two data streams through a single DSP48 slice.

DSP48 Slice

The Virtex-4 DSP system elements are dedicated, diffused silicon with dedicated, high-speed routing. Each is configurable as: an 18 bit x 18 bit multiplier, a multiplier followed by a 48-bit accumulator (MACC), or a multiplier followed by an adder/subtracter. Built-in pipeline stages provide enhanced performance for throughput of 500 MHz. Even though all Virtex-4 devices have DSP48 slices, the SX family contains the largest ratio of DSP48 slices to logic elements. An industry high, the 512 DSP48 slices are ideal for math-intensive applications, such as image processing. The DSP48 slice is also very power efficient due to the use of a triple-oxide, 90 nm process. Architectural features of the DSP48 slice, such as built-in pipeline registers, accumulator, and cascade logic, nearly eliminate the use of general-purpose routing and logic resources for DSP functions and further reduce power. This slashes the DSP power consumption to a fraction of Virtex-II Pro devices.

DDR with Two Data Streams

DDR, as used in this application note, refers to multiplexing two input data streams into one stream at twice the rate and interleaving (in time) the data from each stream as shown in Figure 1. The diagram shows the reverse operation of creating two parallel resultant streams after processing.

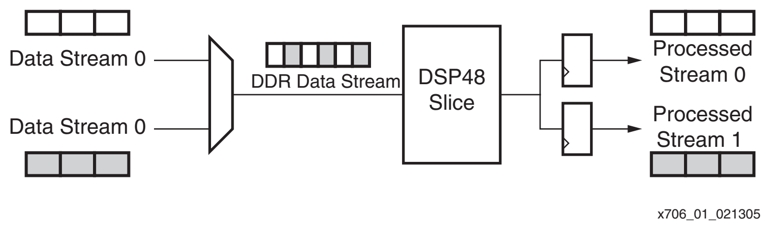

Figure 1: **DSP48 DDR**

The DSP48 slice inputs can be driven at a fast 500 MHz clock rate from CLB flip-flops, CLB LUTs configured as shift registers (SRL16), and directly from block RAM. Block RAM configured as a FIFO using the built-in FIFO support, also supports the 500 MHz clock rate.

Design Considerations

Dealing with data at 500 MHz requires strict pipelining with registers on the outputs of each math or logic stage. The DSP48 slice provides optional pipeline registers on the input ports, on the multiplier output, and on the output port from the adder/subtracter/accumulator. Block RAM also has an optional output register for efficient pipelining when interfaced to the DSP48 slice. Where CLBs are used, only minimal levels of logic should be placed between registers to provide maximum speed. For DDR operation, a 2:1 mux (a single LUT level) is the only required logic between pipeline stages. Whether interfacing to the DSP48 slice with memory or CLBs, place the 500 MHz elements in close proximity to minimize the connection lengths in the general routing matrix.

DDR requires the DSP48 slice operating at double the frequency of the input data streams. A digital clock manager (DCM) provides a phase-aligned double-frequency clock using the CLK2X output.

Another aspect of inserting DDR data through a section of a pipeline is insuring data passes cleanly between clock domains. This can require adding extra registers clocked with the double-frequency clock at the output of the double-pumped section to synchronize the data with the original clock. Typically, to insert a DDR section cleanly into a pipeline, there must be an even number of register delays in the DDR section.

Implementation

There are several configuration options for implementing this functionality. Figure 2 shows a straightforward implementation.

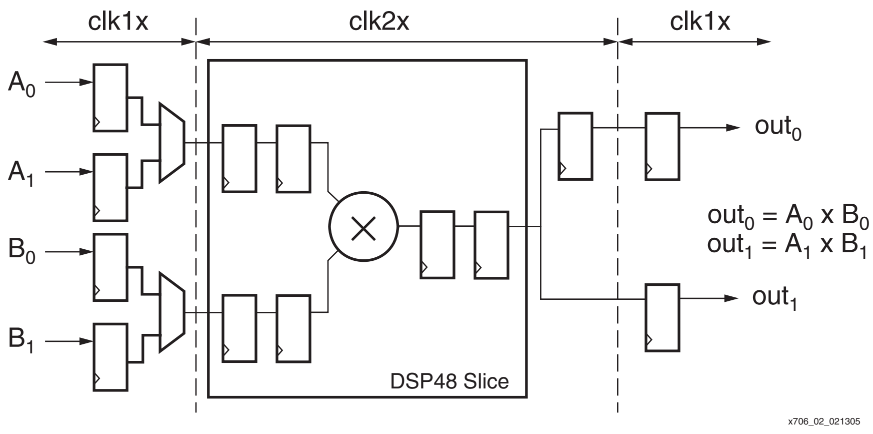

Figure 2: **Two Stream Multiply Through a DSP48 Slice**

In Figure 2, stream 0 consists of A_0 and B_0 inputs. They are multiplied together and output as *out0*. Similarly, stream 1 consists of inputs A_1 and B_1 multiplied together and output as *out1*. There are two clock domains: the *clk1x* domain, at the nominal data stream frequency, and the *clk2x* domain, at twice the nominal frequency. Figure 2 shows two registers after the multiplier. The second is the accumulation register, even though accumulation is not used in this configuration. The register, however, is still required to achieve the full, pipelined performance. There are two sets of registers on the inputs of the DSP to make the total delay through the DSP48 slice an even number (4), providing easier alignment of the output data with *clk1x*. These registers are *free* since they are built into the DSP48 slice, and using them reduces the need for alignment registers external to the DSP48 slice. The extra pipeline register on *out0* compensates for taking stream 0 into the DSP one *clk2x* cycle before stream 1. As shown in the timing diagram in Figure 3, the extra pipeline register on *out0* is required to realign the stream 0 data back into the *clk1x* domain.

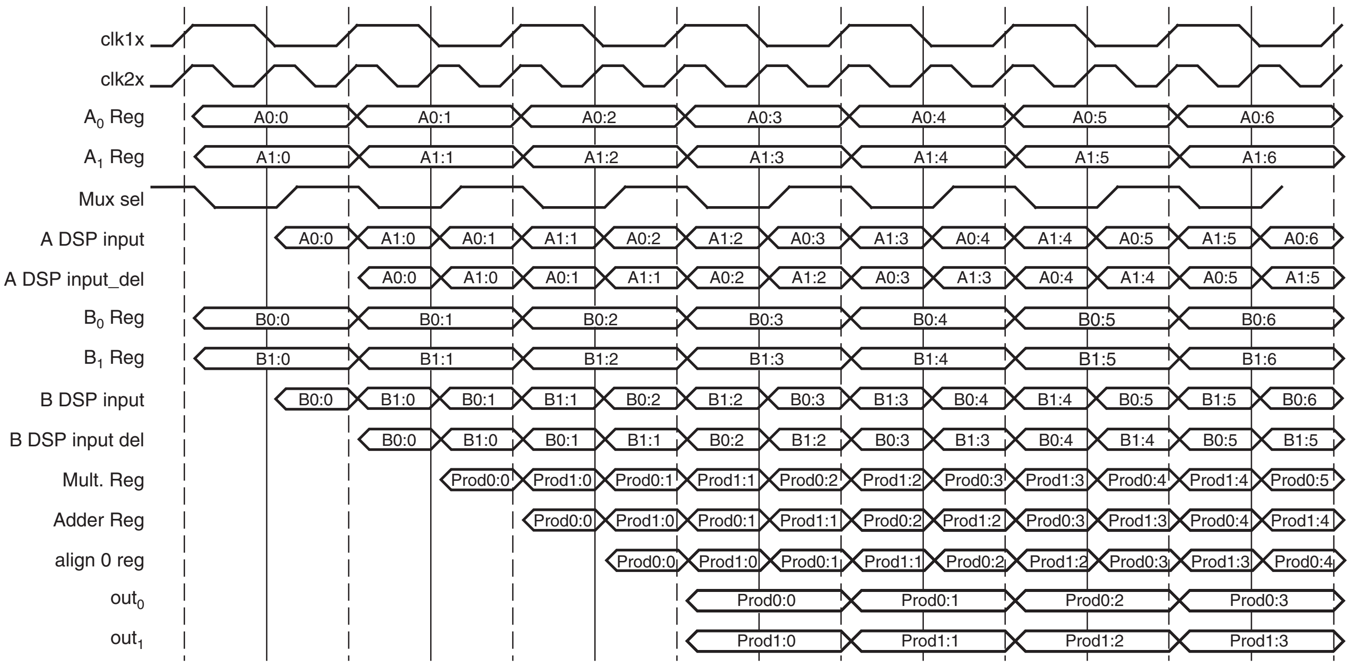
clk1x
clk2x
A0 Reg
A1 Reg
Mux sel
A DSP input
A DSP input_del
B0 Reg
B1 Reg
B DSP input
B DSP input del
Mult. Reg
Adder Reg
align 0 reg
out0
out1
A0:0 A0:1 A0:2 A0:3 A0:4 A0:5 A0:6
A1:0 A1:1 A1:2 A1:3 A1:4 A1:5 A1:6
B0:0 B0:1 B0:2 B0:3 B0:4 B0:5 B0:6
B1:0 B1:1 B1:2 B1:3 B1:4 B1:5 B1:6
Prod0:0 Prod1:0 Prod0:1 Prod1:1 Prod0:2 Prod1:2 Prod0:3 Prod1:3 Prod0:4 Prod1:4 Prod0:5
x706_03_021305

Figure 3: **Timing of a Two Stream Multiply**

The input mux select *mux_sel* is essentially the inverse of *clk1x*. It is important, however, to generate this signal from a register clocked by *clk2x* rather than deriving it directly from *clk1x* to avoid hold-time violations on the receiving registers. To generate the *mux_sel* signal, as well as the control for the accumulator in the DSP48, it is useful to have a *clock follower* signal: a signal with the same pattern as the clock but slightly delayed to eliminate hold problems. This *clock follower* signal is useful for signals with DDR functionality. There are many ways to create this mundane but important signal. One circuit, depending only on the clocks, with minimal and controlled delay, is shown in Figure 4.

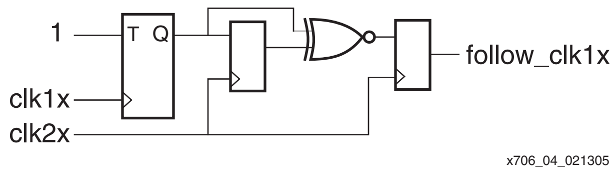

Figure 4: **Clock Follower Circuit**

The clock follower circuit creates a two phase version of *clk1x/2* and compares the two phases. It determines if the next edge of *clk1x* is rising or falling. The timing for this circuit is illustrated in Figure 5. This circuit is part of the reference design.

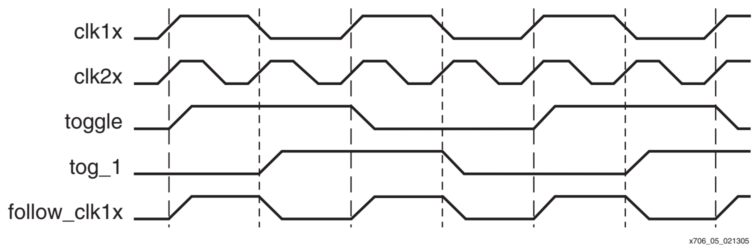

Figure 5: **Clock Follower Waveforms**

As shown in Figure 5, the *toggle* signal (and consequently *tog_1*) can be inverted without affecting *follow_clk1x*. The output is *clk1x* delayed by the *clk-to-q* time of a flip-flop.

At the transitions between clock domains, the data has only one *clk2x* period to set up. This is the reason for having no logical operations between registers in the two domains and why the placement of the first registers in the *clk1x* domain is more critical than other registers in the same domain.

Alpha Blending

Alpha blending of video streams is a method of blending two images into a single combined image, for example: fading between two images; overlaying anti-aliased or semi-transparent graphics over an image; or making a transition band between two images on a split-screen or wipe. Alpha refers to a weighting factor defining the percentage of each image in the combined output picture. For two input pixels, P_0, P_1, and a blend factor, α, where $0 \le \alpha \le 1.0$, the output pixel P_f will be Equation 1:

$$P_f = \alpha P_0 + (1 - \alpha) P_1 \qquad \textit{Equation 1}$$

Figure 6 In graphical terms:

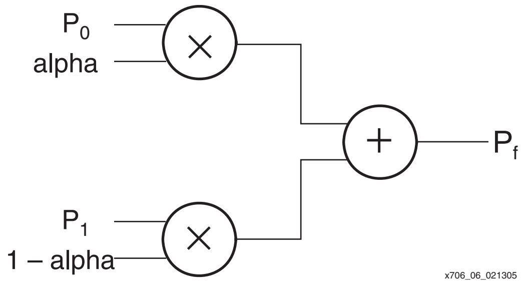

Figure 6: **Alpha Blend**

This operation is performed separately for each component. In this implementation the components are Red, Green, and Blue.

A pixel rate of 250 MHz or less is sufficient for all standard-definition and high-definition video rates, and common VESA standards up to 1600 x 1200 at 85 Hz. Therefore, one DSP48 slice can perform the multiply add on one component. Also, as shown in Figure 7, a set of three slices can alpha blend the three components from each of two video streams. In the remaining discussion, only one color component is shown, but it is understood the operations must be done identically and in parallel on each of the three components. The same technique works for other linear video components such as Y, Cb, Cr.

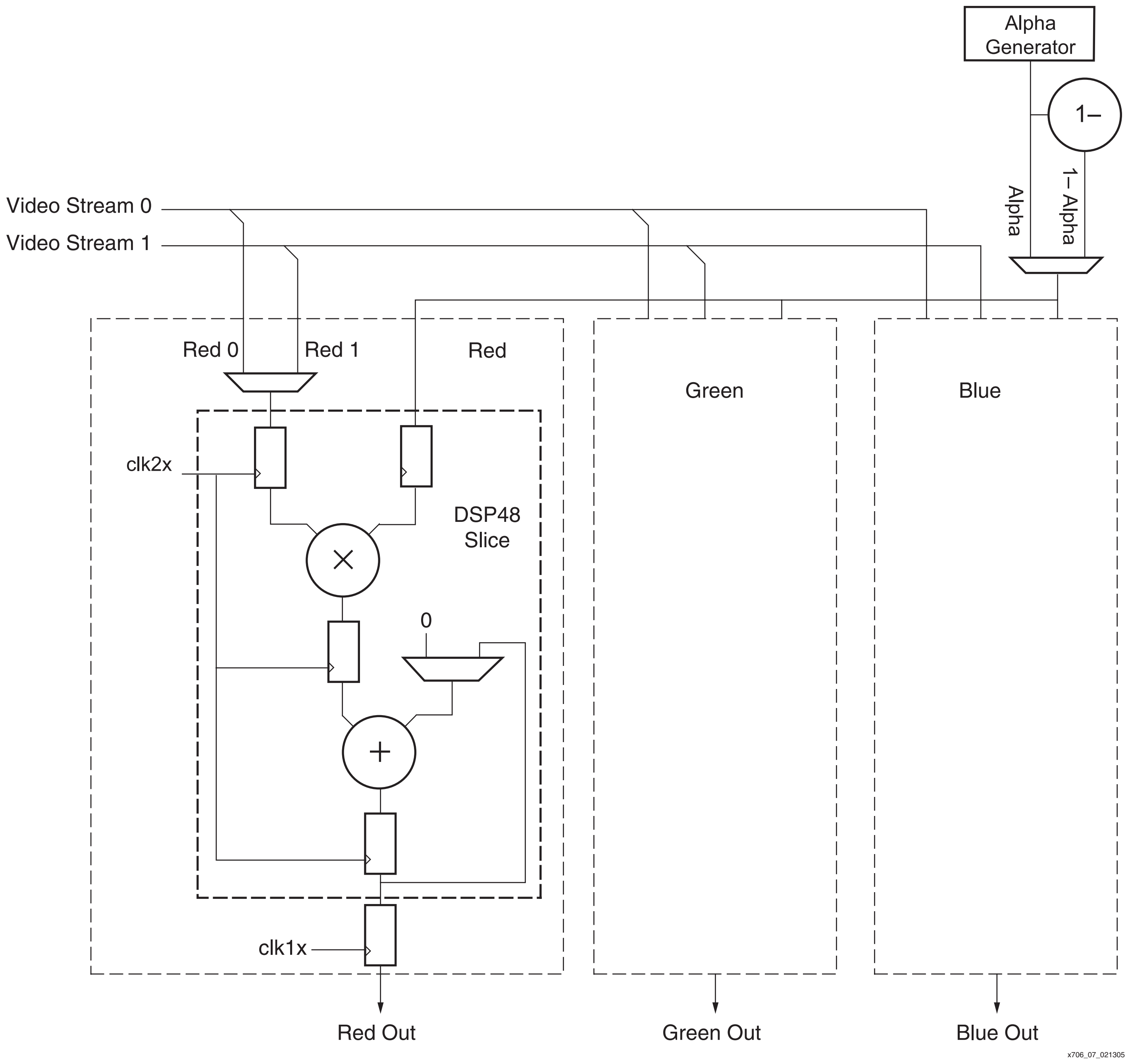

Figure 7: **Alpha Blend on 3-Component Video**

There are several ways to implement alpha blending depending on the nature of the video streams and how alpha is generated. Figure 8 shows a basic implementation with two video streams alternating as one multiplier input. The other multiplier input alternates between *alpha* and *1 – alpha*.

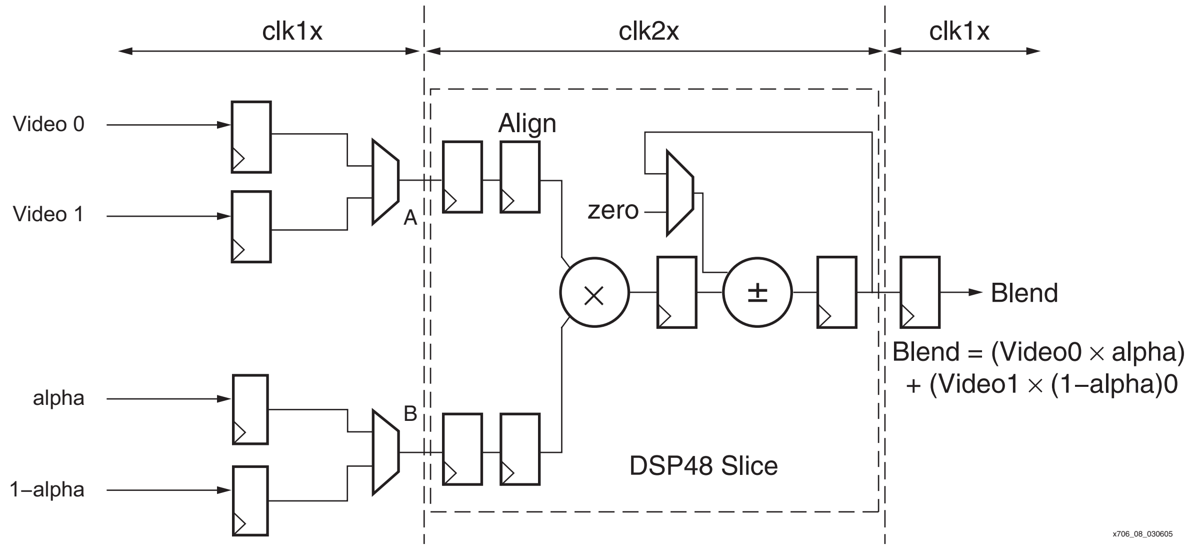

Figure 8: **Alpha Blend Implementation (One Component)**

The operating mode of the adder alternates between *add 0* (pass through) mode and *add output* (accumulate) mode. The DSP48 slice output register contains the result of the *Video0* × *alpha* multiply during one clock cycle, and the final result, *Video1* × *(1 – alpha) + Video0* × *alpha*, on the alternate clock.

Figure 9 shows the timing for this configuration. The *align* registers on the inputs of the DSP are used to make the total delay through the DSP48 slice an even number (4), as explained in "Implementation," page 2. The final output register for *blend* loads new data every other DSP clock to register the blend results at the original pixel rate.

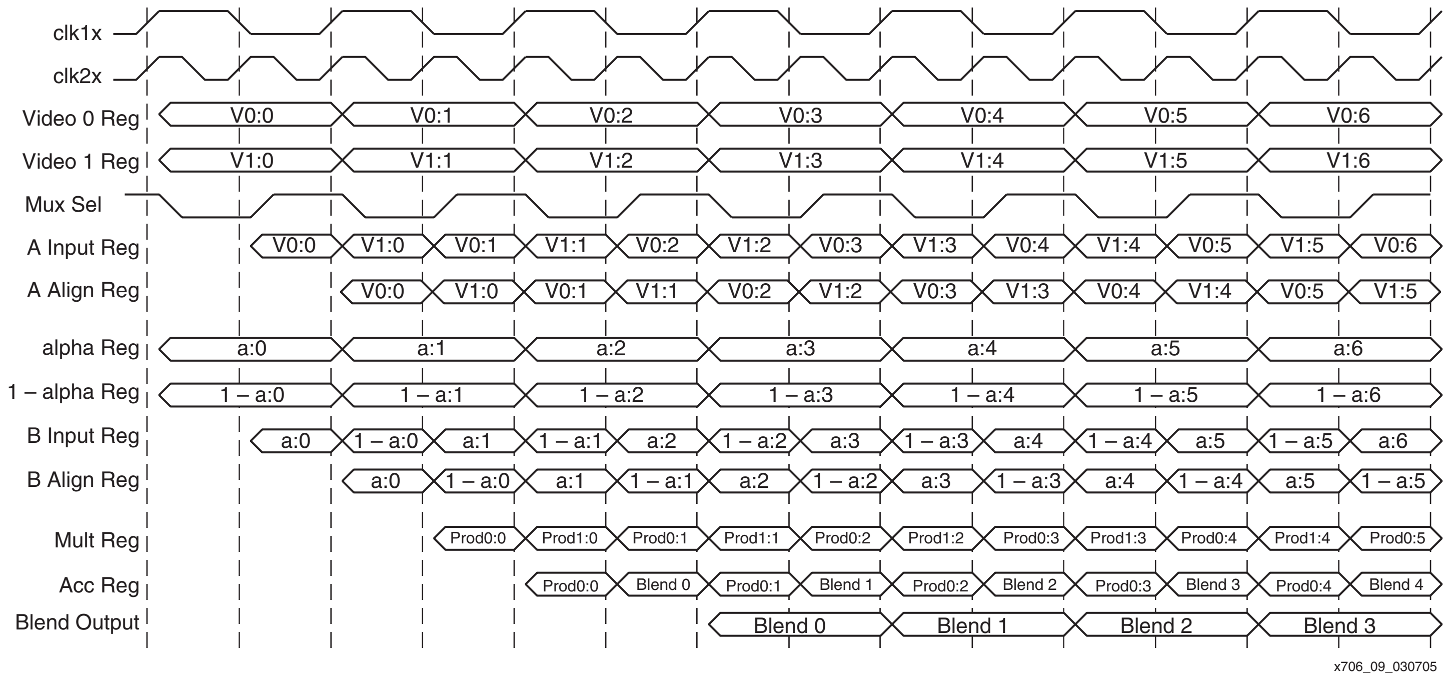
clk1x
clk2x
Video 0 Reg
Video 1 Reg
Mux Sel
A Input Reg
A Align Reg
alpha Reg
1 – alpha Reg
B Input Reg
B Align Reg
Mult Reg
Acc Reg
Blend Output
V0:0 V0:1 V0:2 V0:3 V0:4 V0:5 V0:6
V1:0 V1:1 V1:2 V1:3 V1:4 V1:5 V1:6
a:0 a:1 a:2 a:3 a:4 a:5 a:6
1 – a:0 1 – a:1 1 – a:2 1 – a:3 1 – a:4 1 – a:5 1 – a:6
Prod0:0 Prod1:0 Prod0:1 Prod1:1 Prod0:2 Prod1:2 Prod0:3 Prod1:3 Prod0:4 Prod1:4 Prod0:5
Prod0:0 Blend 0 Prod0:1 Blend 1 Prod0:2 Blend 2 Prod0:3 Blend 3 Prod0:4 Blend 4
Blend 0 Blend 1 Blend 2 Blend 3
x706_09_030705

Figure 9: **Alpha Blend Timing**

Equalizing Delays in Video Streams

Block RAM can be used to source the video data and/or the blend factors. Use this capability, for example, to equalize delay between two video streams. If two video streams are received from different sources, the first pixel of the first line might not be available from both streams (i.e., the two streams are not perfectly pixel aligned), even though the pixel rate is the same. To merge with an alpha blend, the pixels must be perfectly aligned and have a common clock. If the pixel misalignment is less than the number of pixels in a few scan lines, the built-in Virtex-4 FIFO logic can be used to delay the leading stream so corresponding pixels from both streams go into the DSP48 slice at the same time. For pixel misalignment extending to a large portion of a frame, it can be more cost effective to use an external frame buffer. Using a FIFO also facilitates the synchronization of video streams from different clock domains by clocking the input of the FIFO in a different clock domain than the output of the FIFO. Figure 10 illustrates this configuration.

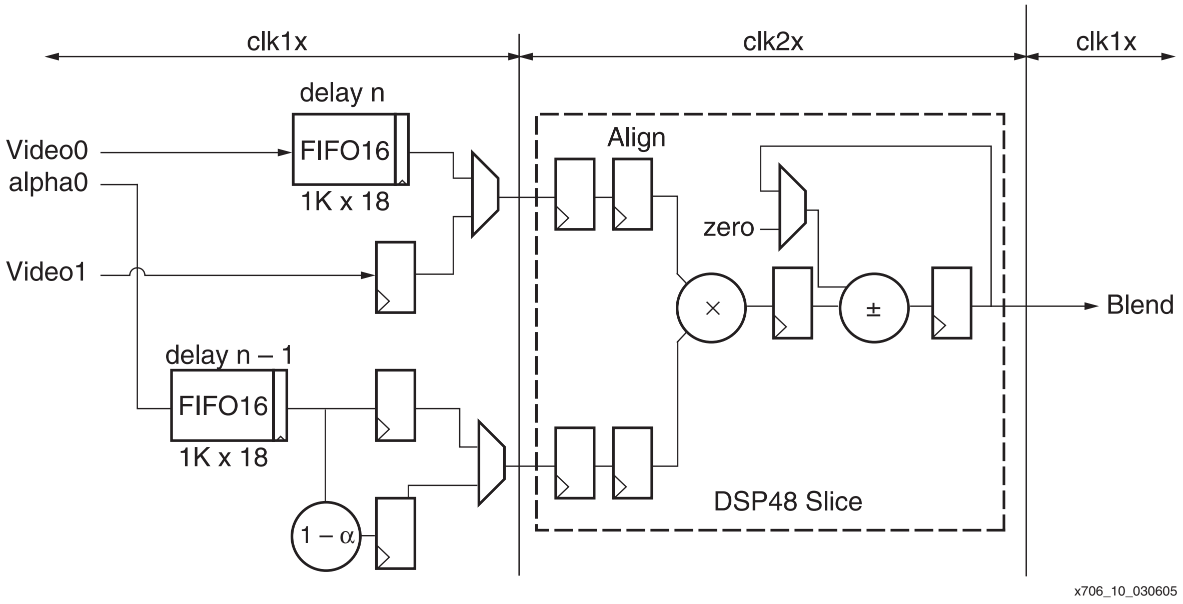

Figure 10: **Video Stream Delay Equalization**

In this example, the *Video0* stream including *alpha0* is advanced in time from video stream 1, *Video1*, by n *clk1x* periods. The FIFO on *Video0* delays the pixel data to match *Video1*. The FIFO on *alpha0* delays alpha by n – 1 clocks. The *1 – alpha* value is created on the output side of the alpha FIFO. Thus, all of the data going into the DSP slice is concurrent, that is, it corresponds to the same pixel in the frame.

This equalizing delay is capable of compensating for timing mismatch to over 1000 pixels for the example shown and up to several lines by using cascaded FIFOs. If the delays required are small (<16) and constant, CLBs configured as a shift register (SRL16) can be used as the delay elements.

Non-Linear Alpha Functions

One case where alpha blending is useful is in performing a wipe between two video streams. To create a *feathered edge,* or blend region, along the border between the two images, alpha blending can be performed across a transition band of a defined width. A linear alpha ramp in the transition region is most easily calculated; however, a non-linear ramp results in the smoothest transition. Figure 11 illustrates the feathered wipe with a transition region.

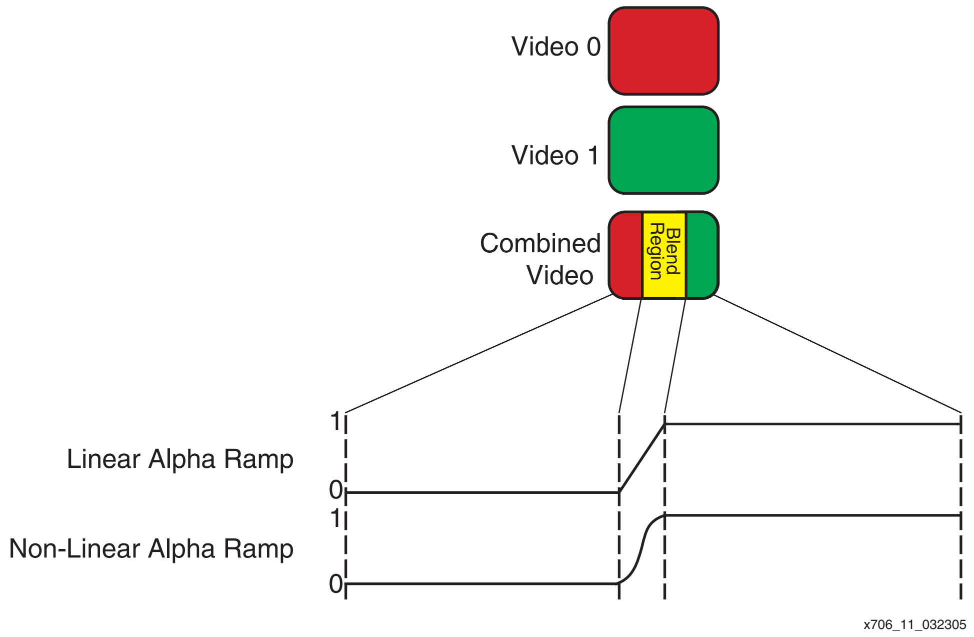

Figure 11: **Feathered Transition Region on a Horizontal Wipe**

Conveniently, a block RAM can be used as a look-up table to convert a linear ramp function into an arbitrarily shaped, non-linear function as shown in Figure 12.

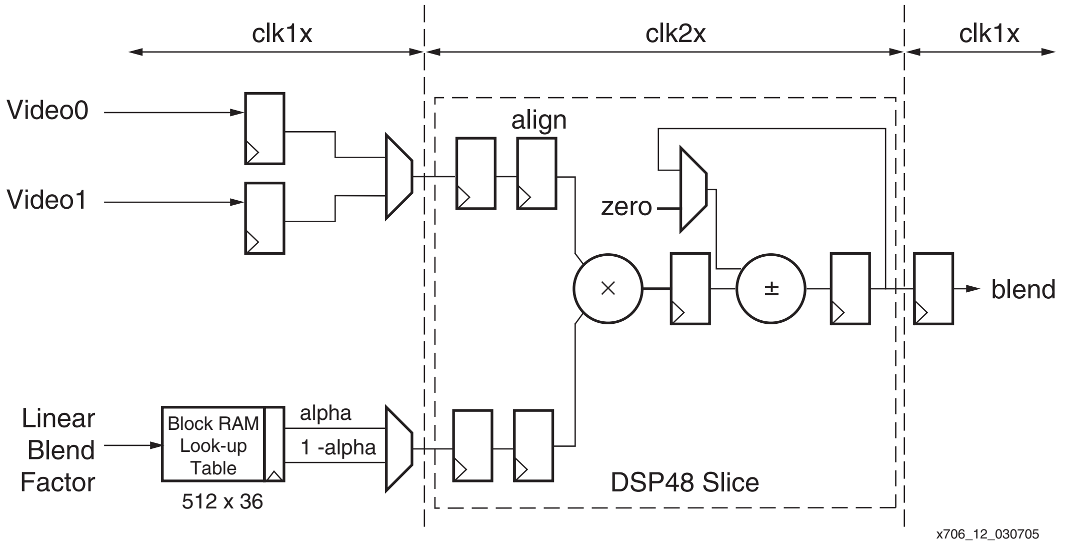

Figure 12: **Non-Linear Blending Using a Block RAM Look-Up Table**

This example, converts a 9-bit blend factor to an 18-bit *alpha* and an 18-bit *1 – alpha*. Figure 13 illustrates the smooth-feathering LUT function.

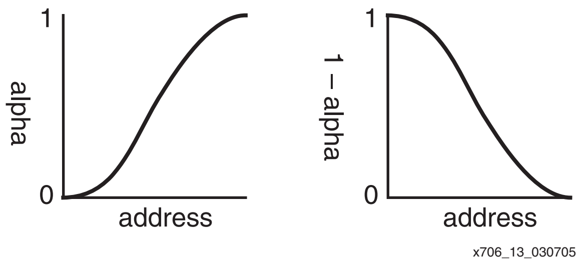

Figure 13: **Non-Linear Blend Function**

Although strictly speaking, alpha blending requires scaling components sum to 1, interesting results are obtained by combinations in which the output value sum to less than 1. For example, the same linear ramp input to the LUT can produce a transition from *Video0* to black to *Video1* using the functions illustrated in Figure 14.

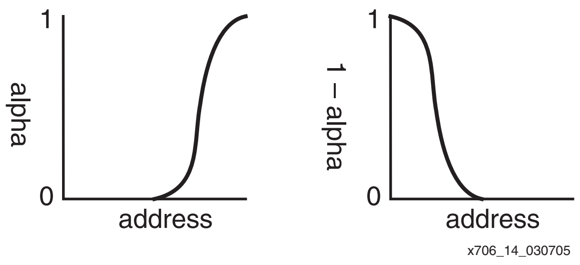

Figure 14: **Fade-to-Black-Border Blend Function**

Avoid combinations where the output values (*alpha* + *"1 – alpha"*) sum to more than 1 to avoid saturation, color wrap, and/or color shift artifacts.

Reference Design

The reference design is an implementation of the basic alpha blend shown in Figure 8, page 6. The video stream in and out consist of 10 bits each of red, green, and blue, plus HSYNC, VSYNC, and data enable. The alpha input is also 10 bits. The *dual_stream_blend* module containing the DSP48 HDL code has ports for the full 18 bits in and 48 bits out of the DSP48 slice. Synthesis eliminates the unused registers and logic.

The implementation consists of two modules, *dual_stream_blend* and *alpha_blend_top* as shown in Figure 15. The *dual_stream_blend* modules allow access to the full 18-bit inputs and 48-bit outputs of the DSP48 slice. *Alpha_ blend_ top* instantiates the red, green, and blue blenders and a DCM to produce the *clk2x.* It also handles converting the 10-bit inputs and outputs to the DSP48-sized inputs and outputs.

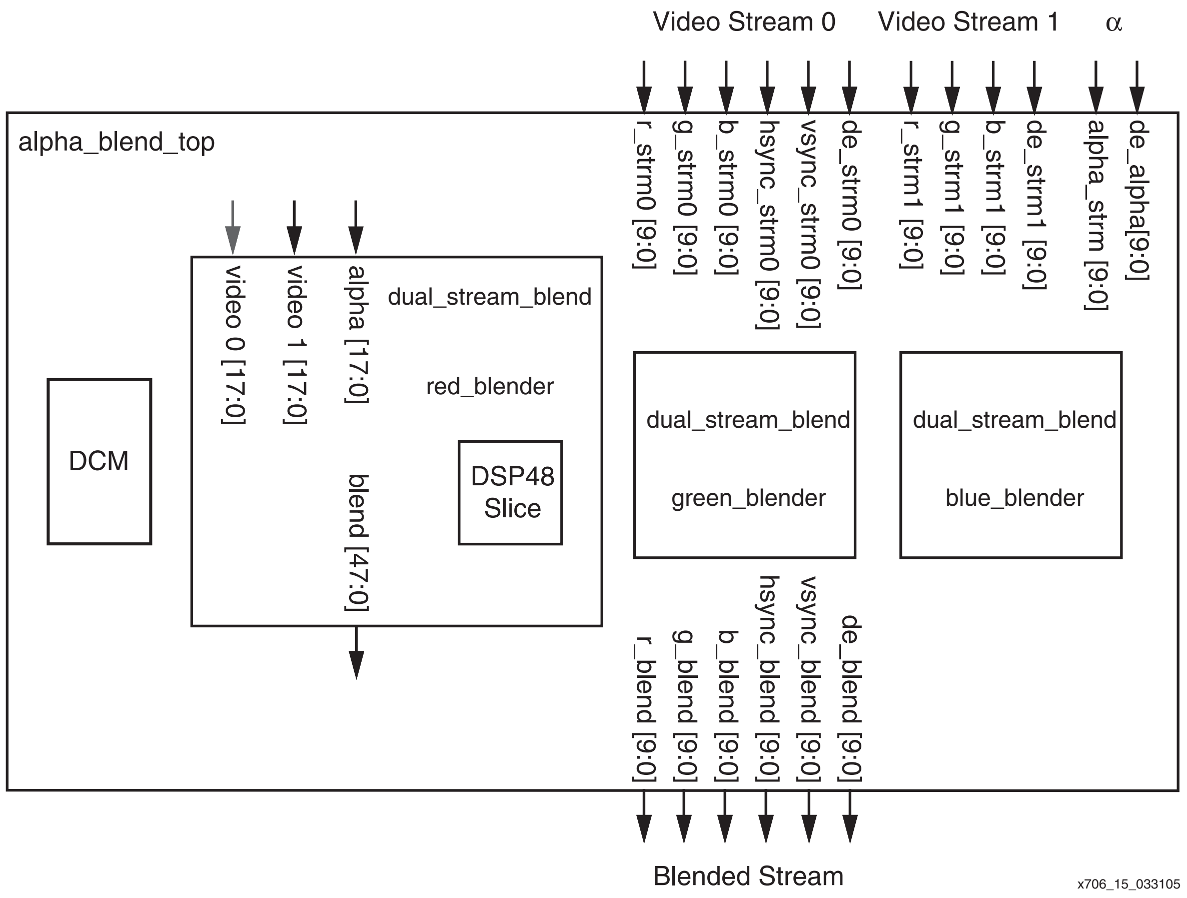

Figure 15: **Reference Design I/O and Hierarchy**

Bit replication, repeating MSBs in the low order bits of the inputs to the DSP, is used to preserve range at high values. Rounding the final result to 17 significant bits is used to preserve accuracy at the low end of the scale (see Appendix A: Bit Replication and Rounding). The bit mapping used in the reference design, including bit replication and rounding, is shown in Figure 16.

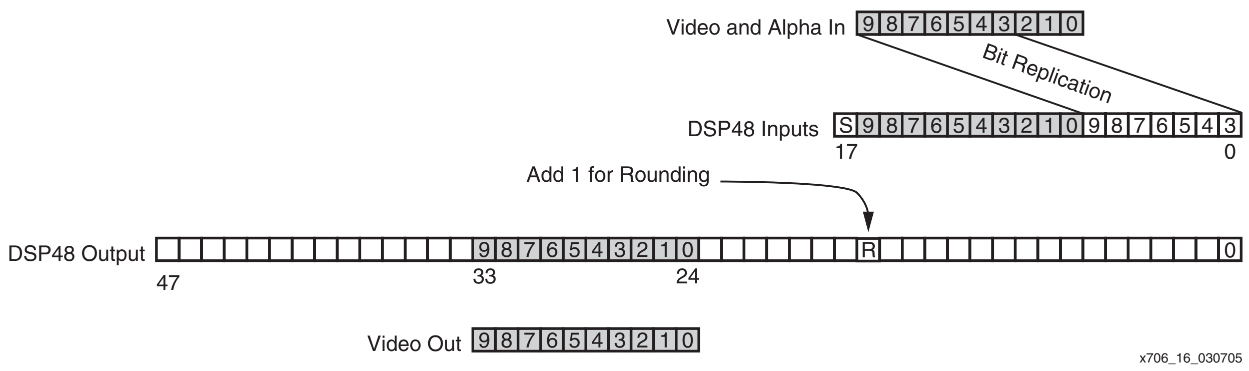

Figure 16: **Bit Mapping in the Reference Design**

Neither the bit-replication nor the rounding consume any general logic resources; both simply make use of resources available in the DSP48 slice.

Reference Design Results

Table 1 shows the results, after place and route, of the modules implemented in this application note. These results were obtained using the VHDL versions with Xilinx ISE version 7.1i and XSE synthesis. Results using the Verilog files are not shown but are essentially identical. The results are for a Virtex-4 device with a -12 speed grade.

Table 1: **Reference Design Results**

Speed (clk2x)	FF	LUT	DCM	BUFG	DSP48
500 MHz	121	64	1	2	3

Conclusion

The high performance of Virtex-4 with DSP48 slices is efficiently used by processing multiple data streams in a time-multiplexed fashion. With careful design, a single DSP48 can perform multiply operations on two independent data streams, operating at 250 MHz each. Alpha blending of video streams, as provided in the reference design, is one example of processing two data streams through a single DSP48 slice.

Appendix A

Bit Replication and Rounding

Bit replication for multiplication is loosely analogous to rounding. The purpose, in both cases, is to reduce quantization errors. While rounding adds ½ LSB to the result before truncation, bit replication scales each factor by (1 + ½ LSB) before the product is calculated. Bit replication is practical when the width of the multiplier is greater than the width of the incoming factors, as is the case in the reference design. For optimal results of alpha blending, bit replication and rounding can be used together.

One way to evaluate the accuracy of the alpha blend calculations is to calculate the output when the value of both inputs is identical. Consider the condition when both video streams have the same value, n. Under such conditions, the output should be Equation 2:

$$(alpha \times n) + (1 - alpha) \times n = n \qquad \textit{Equation 2}$$

That is, the output should match the input value regardless of the value of alpha. Figure 17 shows the calculated outputs for several rounding and bit replication conditions. These results are explained in the following sections.

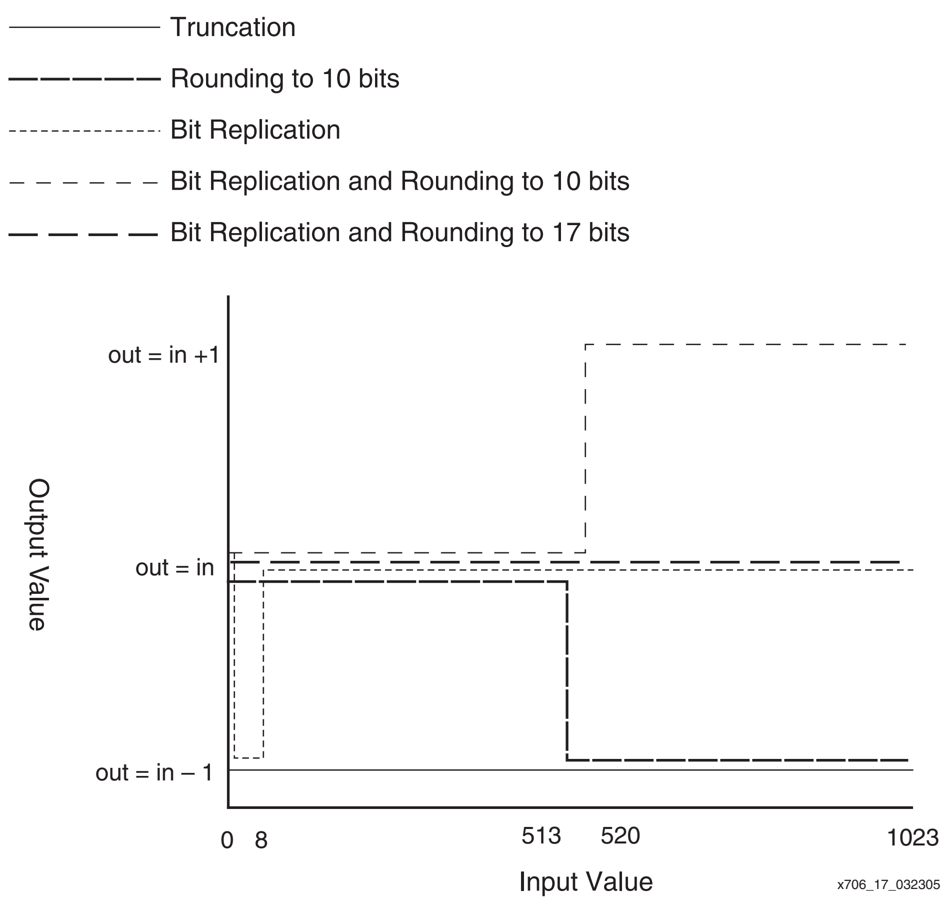

Figure 17: **Output vs. Input for Identical Inputs**

In all cases, the output never differs from the ideal by more than one. This difference of one may not be significant in itself; however, if there are multiple stages of processing, the small loss of accuracy at each stage can add up to significant and visible errors in the final result.

Rounding

Simple truncation results in an output that is always one less than the input. Since the reference design uses a 10-bit output, it is logical to round to 10 bits by adding ½ of the LSB (see Reference Item 2, page 14.) Indeed, this corrects the output for the lower half of the range, but the top half of the range is still reduced by one. This is where bit replication can help.

Bit Replication

Bit-Replication Method For Up-Multiplying (Reference Item 1, page 14) contains the theoretical and mathematical treatment of bit replication. It shows how and why bit replication should be used when values need to be converted from a lower precision range to a higher precision range. For the purposes of the reference design of this application note, it is not necessary to convert to a higher precision range; the output resolution is the same as the input resolution. Bit replication does, however, offer important computational advantages as evidenced by the *Bit Replication* line in Figure 17.

In video applications, the component and alpha values represent a normalized number between 0 and 1 with an implied binary point to the left of the MSB. Thus, in practice, 1 is represented by number with binary 1s in all bits. For an n-bit number, the highest value that can be represented is $(2^n - 1)/ 2^n$. Likewise, for computational simplicity, the *1 – alpha* value is derived by the 1's complement of *alpha* (also having a range 0 to all 1s). The downside of this is the multiplication of the component values by *alpha* or *1 – alpha* tends to compress the range of output values. For example, the 8-bit result of FF × FF is FE.

Although it cannot add new information, bit replication helps preserve the information available. As shown in Figure 18, it scales the value of the codes such that the highest value is much closer to1 for the purposes of multiplication, just as 0.99 is a better approximation of 1 than 0.9, an 8-bit FF Hex is a better approximation than a 4-bit F Hex. The implementation of scaling is trivial; it is a matter of routing input bits to more than one place. In the reference design, the extra precision required is free when using the18-bit input ports and 48-bit output port of the DSP48 slices.

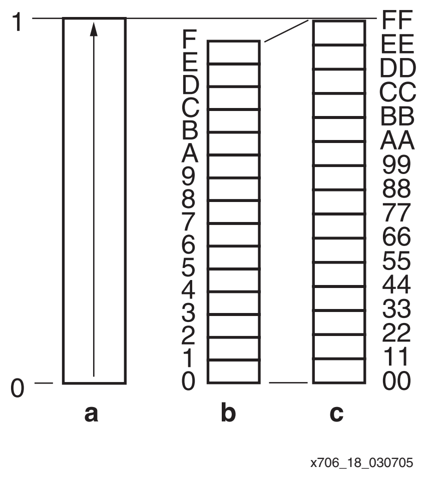

Figure 18: **Number Space Quantization**

In Figure 18:

- "a" is a continuous range 0 - 1
- "b" is 4-bit quantization
- "c" is 4-bit quantization, bit replicated to 8-bits.

A simple example illuminates how bit replication works. Suppose two 4-bit data streams are multiplied in a multiplier with 8-bit inputs. The 4-bit values range from 0 to 15 (F Hex). Assuming these are video values representing fractions of the scale 0 to 1, the highest value that can be represented is 15/16. Of course, the factors could be input to the four LSBs of the multiplier; however, this results in a maximum product of E1 Hex. After quantization, E1 Hex becomes E Hex, representing 14/16. If this output were subsequently multiplied by another 4-bit value, the maximum product would be D2 (after quantization, D or 13/16). The accumulation of error and corresponding compression in the range of outputs is apparent. Using rounding to add 8 to the product, the quantized value does not change.

Bit replication from 4 bits to 8 bits (simply repeating the 4 bits as both MSBs and LSBs of the 8-bit value as shown in Figure 19) has the effect of scaling the factors by (1 + 1/32.) This makes the maximum value of the input data (F Hex) represent 255/256, a value much closer to 1, illustrated as "c" in Figure 18. Each number in the range is slightly offset and expanded from the non-bit-replicated range, "b" in Figure 18.

This, in turn, means the output range is also expanded. The maximum product of the bit-replicated factors is FE01 Hex (normalizes to the full F Hex). Using bit replication, further multiplications do not compress the range of possible products.

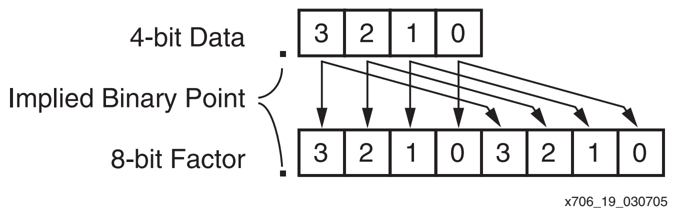

Figure 19: **Bit Replication From 4 Bits to 8 Bits**

Referring again to the reference design, notice in Figure 17, with bit replication, the output is accurate at all input values except 1 to 7. This is because, for these values, bit replication has no effect on the inputs (refer to Figure 16). When the values are this small, the relative error is great. For example, when a value of 1 is expected, a 0 is received. The percentage error is infinite.

Bit Replication with Rounding

Since, rounding to 10 bits corrects all of the lower values, it might seem like this should be done in addition to bit replication; however, this leads to an error of +1 at the high end of the range. This is catastrophic at the maximum input value of 1023, because the output value of 1024 exceeds the range of the output field and, in practice, is quantized as 0.

The key to using rounding is to understand that with bit replication, there are now 17 significant bits in the result. Thus, by rounding the results of the multiply and add to 17 bits, the very low values (1 to 7) are corrected, without affecting values at the high end. (See Figure 17 “Output vs. Input for Identical Inputs”.) Thus, the reference design uses bit replication to 17 bits on the input factors before the multiply, and rounding to 17 bits on the final add (Figure 16) to improve accuracy and preserve the output range at all values. This is done by utilizing the resources of the DSP48 slice and requires no additional logic.

References

1. *Bit-Replication Method For Up-Multiplying* by Robert A Ulichney and Shiufun Cheung, Digital Equipment Corporation, Cambridge Research Laboratory, January, 1997.
2. *XtremeDSP Design Considerations User Guide*, Chapter 1, page 32, "Symmetric Rounding Supported by Carry Logic", Xilinx Inc., September 2004.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
03/31/05	1.0	Initial Xilinx release.