

Distributed Arithmetic Laplacian Filter

A common practice in image processing involves convolving an image with a Laplacian operator. **Figure 1** shows a

-16	-7	-13	-7	-16
-7	-1	12	-1	-7
-13	12	160	12	-13
-7	-1	12	-1	-7
-16	-7	-13	-7	-16

Figure 1 - Example Laplacian operator for edge enhancement in an image processing system.

typical Laplacian operator that might be used for edge enhancement. To convolve it with an image, the operator is moved over the image, and centered over each pixel in turn. In each position, the 25 weights in the matrix are multiplied and accumulated with the 25 pixels that the matrix covers. This operation yields one pixel in the resulting image.

This is an ideal application for “distributed arithmetic” techniques that exploit the lookup-table (LUT) architecture of the XC4000E™ FPGA family. **Figure 2** shows the basic approach. Four external line buffers plus the incoming video data provide simultaneous access to five lines of the image. Inside the FPGA, each of the video streams is serialized and passed

through four 1-bit-wide shift registers, each of which delay the data by one pixel. This provides simultaneous bit-serial access to five adjacent pixels from five adjacent lines — the region covered by the Laplacian filter. The shift registers can be implemented very efficiently using the CLB RAM feature of the XC4000E FPGA architecture.

In the most basic distributed arithmetic approach, the 25 signals address a 2^{25} -word LUT which, in turn, feeds a shifting accumulator. This is obviously impractical. A typical cost-reduction measure would be to partition the problem, segmenting the addresses into multiple smaller LUTs. The outputs of these smaller LUTs would be combined in an adder tree to provide the input to the accumulator.

In this particular case, however, the weighting values involved permit the use of more efficient techniques. Except for the values 160 and -7, each of the coefficients is used in four places.

FIFO Buffer

Continued from previous page

TION must start in the reset state when the FIFO is initiated with both counters at zero.

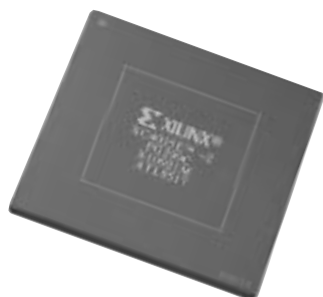
DIRECTION is thus established well before the actual FULL or EMPTY condition can occur. There will be at least four, and usually many more, consecutive set or reset inputs to the DIRECTION latch or flip-flop before it is being used to discriminate between FULL or EMPTY.

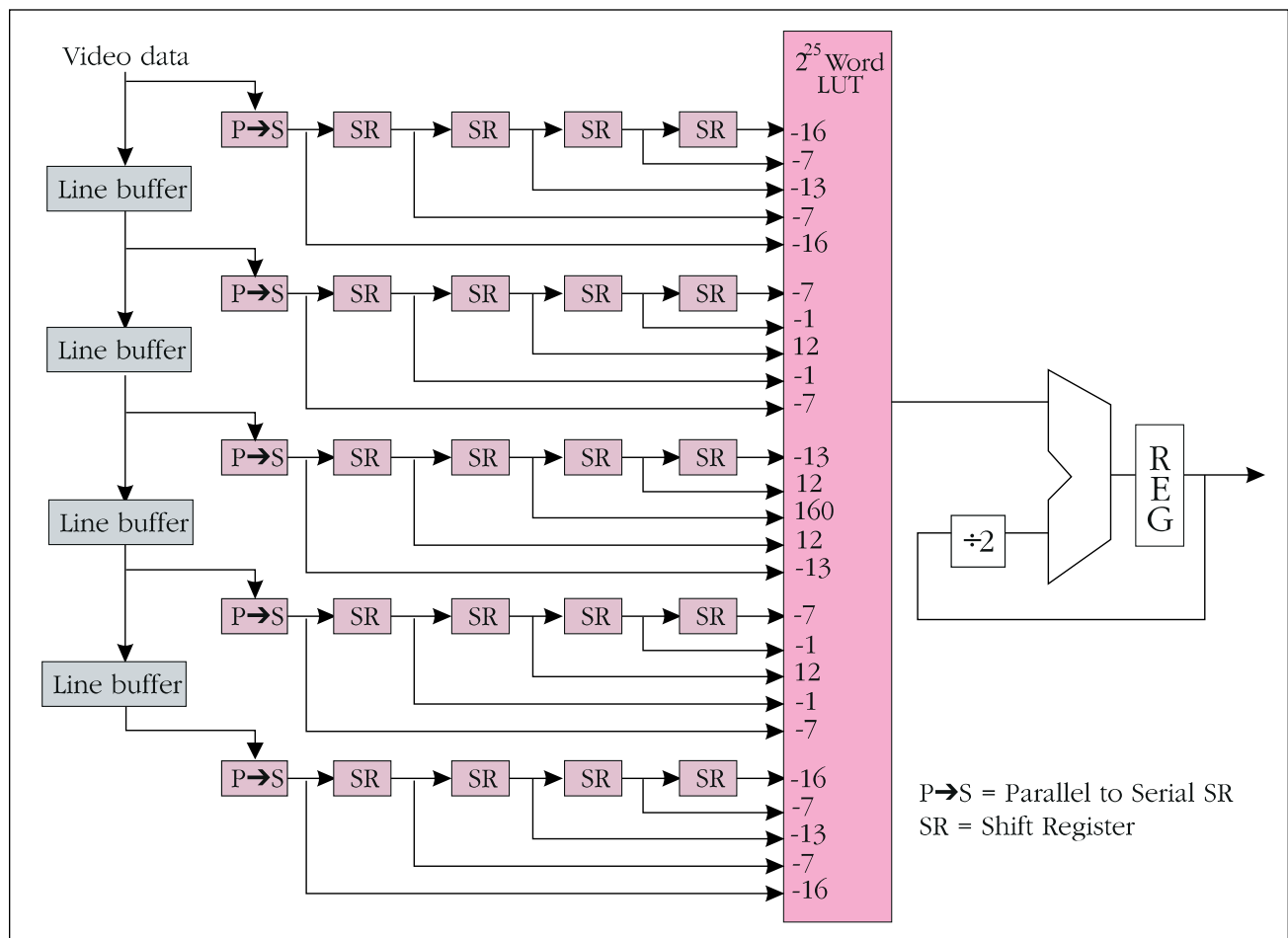
FULL goes active as a result of the write clock edge that writes data into the last available location. FULL goes inactive as a result of the first read clock that reads one word out of the previously full FIFO buffer. EMPTY goes active as a result of the read clock edge that reads the last available data from the FIFO buffer.

EMPTY goes inactive as a result of the first

write clock that writes one word into the previously empty FIFO buffer. In a synchronous design, FULL and EMPTY are synchronous control signals, to be used appropriately by the logic external to the FIFO buffer.

The application note goes on to describe an asynchronous version of the 16x16 FIFO buffer, and 32x8 and 64x8 FIFO buffers with both synchronous and asynchronous read and write clocks. The larger FIFO buffer designs include input and output data multiplexing between multiple RAM banks. The asynchronous 32x8 FIFO buffer requires 28 CLBs and the 64x8 FIFO buffer needs 48 CLBs; both can perform simultaneous read and write operations at 40 MHz. ♦





Consequently, serial adders can be used to combine four serial bit streams into one before addressing the multiplying LUT (Figure 3). Effectively, this adds, and then weights, data that would otherwise be weighted and added afterwards. A tree of three serial adders is needed in each case, and each serial adder can be implemented in a single CLB.

The value -7 is used eight times. The eight inputs could be combined into one, but here it is only necessary to reduce them to two lines. The value 160 is used once, and the data only needs to be delayed to match the delay introduced into the other paths by the serial adders.

These modifications reduce the size of the LUT from 2^{25} words to 2^7 words. This is a large reduction gained from a small amount of logic, but the LUT still is so large that it would have to be split. However, further techniques can be applied.

The -13, -16 and the two -7 values need to be combined into a LUT (Figure 4). The values 12 and 160, however, have no non-zero bit locations in common. Consequently, all possible sums of the two values can be achieved by simply wiring the input signals to appropriate bits of the adder.

All that remains is to handle the -1 value. This value uses the carry input of the adder, but requires some trivial modification to the LUT. Instead of containing all of the possible sums of -13, -16, -7 and -7, it is loaded with all the sums of 13, 16, 7 and 7. The minus signs are accommodated by subtracting in the "adder."

The output of the LUT is inverted as part of a conventional invert-and-add-one method of subtraction. The -1 value can then be accommodated by simply omitting the "add-one" when necessary.

Figure 2 - Basic approach to implementing the Laplacian operator.

*Continued
on next page*

Figure 3 - Serial bit streams can be combined before entering the LUT.

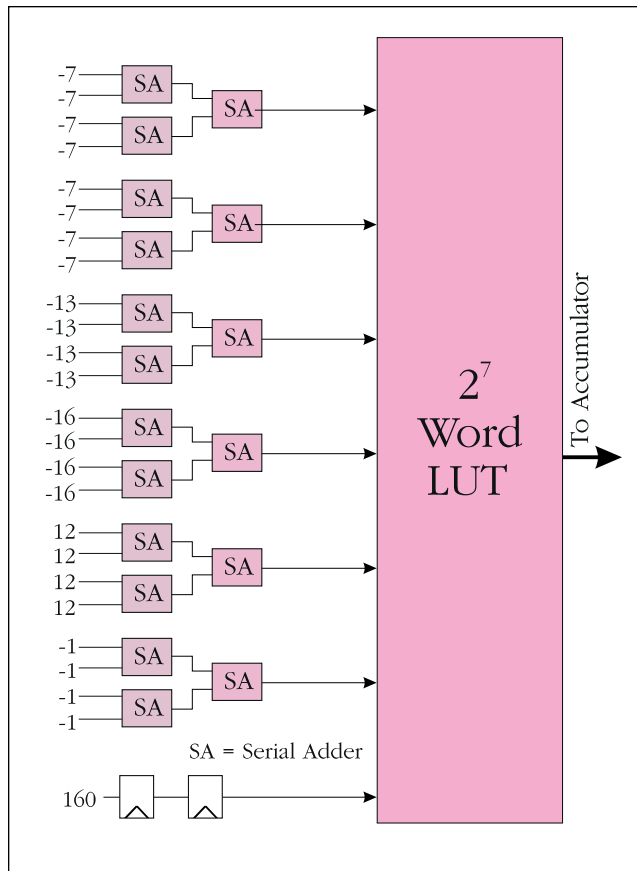


Figure 4 - (top) Final implementation of Laplacian operator.

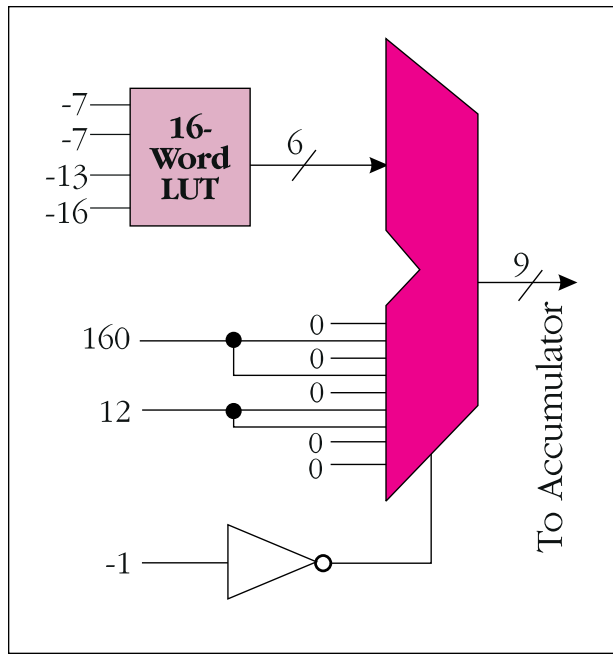
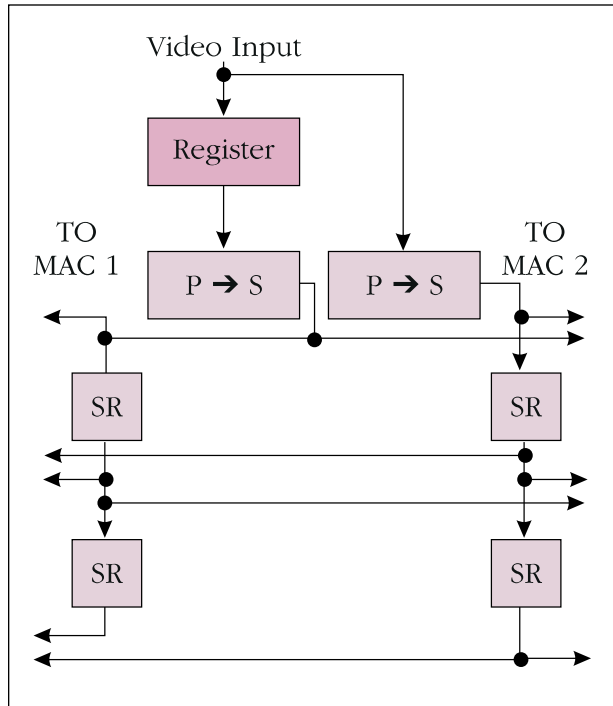


Figure 5 - (right) Sharing input shift registers between parallel MACs.



Laplacian

Continued from previous page

That means the inverted -1 input is used as the carry input to the adder.

In this way, the 2^{25} -word LUT is reduced to a 16-word LUT plus 18 serial adders and one 9-bit parallel adder.

Bit-serial arithmetic inherently involves multiple cycles per pixel. With the multiplier-accumulator reduced to such a small size, however, two or more of them could be used in parallel to regain throughput.

Figure 5 shows how input shift registers can be shared between two MACs.

For each MAC operation, two pixels are brought in, and loaded simultaneously into two parallel-to-serial converters. This requires an additional register to temporarily hold one of the pixels. Six serial outputs are formed, and these are used as two overlapping sets of five each. ♦