



XAPP194 (v.1.0) July 20, 2004

# Serial-to-Parallel Converter

Author: Paul Gigliotti

## Summary

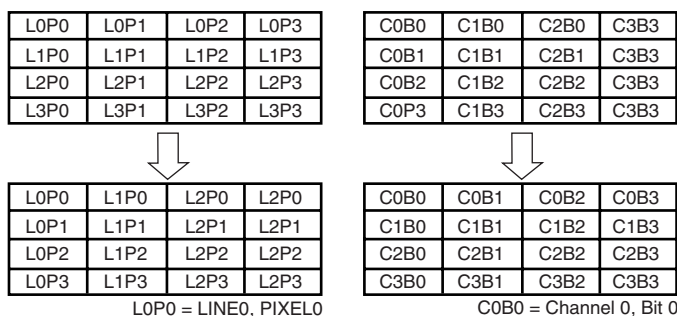
This application note describes the transformation of multiple synchronous serial data streams to parallel data through a multi-channel Serial-to-Parallel Converter. The design, the system architecture, data flows, and design considerations are also discussed.

## Introduction

A previously written article in Issue 31 of the Xcell Quarterly Journal discussed a method for efficiently converting several synchronous serial streams of data to parallel data. That method took advantage of the Virtex/4K distributed RAM to both serialize and double buffer the incoming serial streams. The overall design was efficient, but required the machine to run at eight times the serial rate, which has obvious disadvantages, including limiting the speed of the input ports to a rate eight times slower than the FPGA is capable of attaining. Upon additional study of the problem, the requirement for the eight-times clocking scheme was eliminated and additional bandwidth attained with a reasonable amount of overhead.

### Multi-Channel Serial-to-Parallel Conversion vs. Image Rotation

The transformation of multiple synchronous serial data streams to parallel data is nearly identical to rotating a bit mapped image, as shown in Figure 1. Specifically, columns of data become rows, and vice-versa. Thus, the following serial-to-parallel converter could also be used as a rotation engine for single-bit-per-pixel images. Multiple instances of the design could be run in parallel for multi-bit-per-pixel images.



X194\_01\_011701

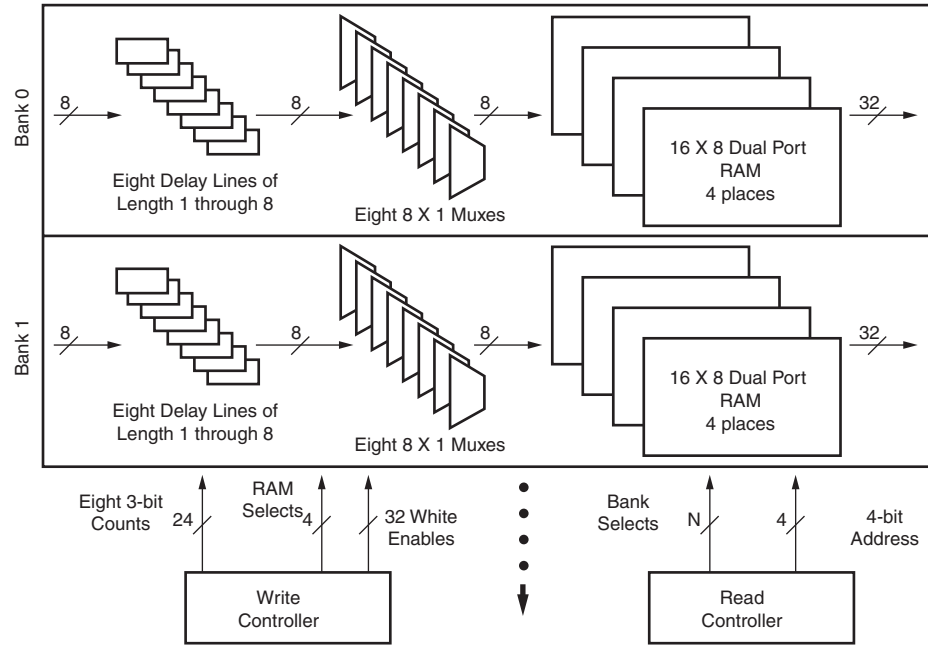
Figure 1: Pixel Rotation vs. Serial-to-Parallel Data Conversion

## Overall System Architecture

In Figure 2, the incoming serial data is grouped into modules of eight. Each module consists of eight delay lines, eight 8X1 muxes, and four 16X8 dual port RAMs. Additional modules can be added to handle additional channels as necessary. The modules share common read and write control circuits. In a traditional shift-register/holding register approach, it would take 64 flip-flops, or 32 Virtex slices per serial channel. This architecture requires only 5 slices per channel, plus the control overhead

© 2004 Xilinx, Inc. All rights reserved. All Xilinx trademarks, registered trademarks, patents, and further disclaimers are as listed at <http://www.xilinx.com/legal.htm>. All other trademarks and registered trademarks are the property of their respective owners. All specifications are subject to change without notice.

NOTICE OF DISCLAIMER: Xilinx is providing this design, code, or information "as is." By providing the design, code, or information as one possible implementation of this feature, application, or standard, Xilinx makes no representation that this implementation is free from any claims of infringement. You are responsible for obtaining any rights you may require for your implementation. Xilinx expressly disclaims any warranty whatsoever with respect to the adequacy of the implementation, including but not limited to any warranties or representations that this implementation is free from claims of infringement and any implied warranties of merchantability or fitness for a particular purpose.

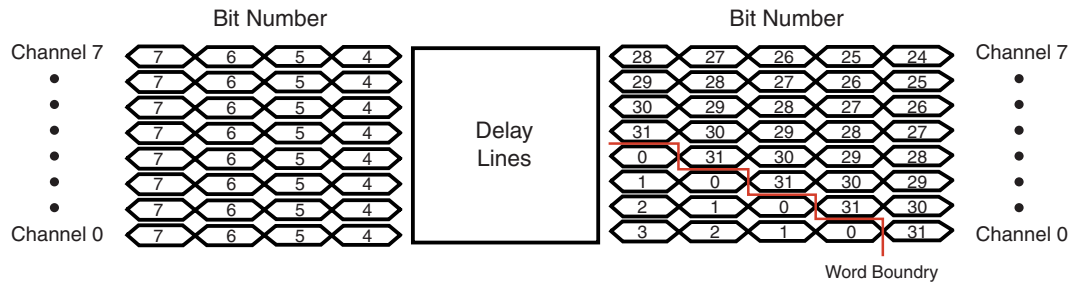


X194\_02\_041404

Figure 2: System Architecture

### Data Flow Through the Delay Lines

The delay lines are constructed out of the Virtex SRL16E, shift register LUTs, which can offer up to 16 times improvement in density when compared to register-based delay lines. The delay lines require only 8 LUTs, which require 2 CLBs. A flip-flop implementation would take 28 flip-flops or 7 CLBs. Each delay line has a different delay length, ranging from one to eight. This causes the incoming synchronous serial data to be skewed one bit position per channel, as shown in Figure 3.



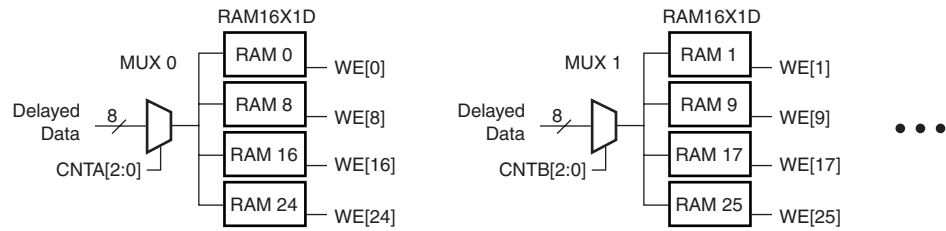
X194\_03\_041404

Figure 3: Delay Line Data Flow

### The Multiplexor Stage

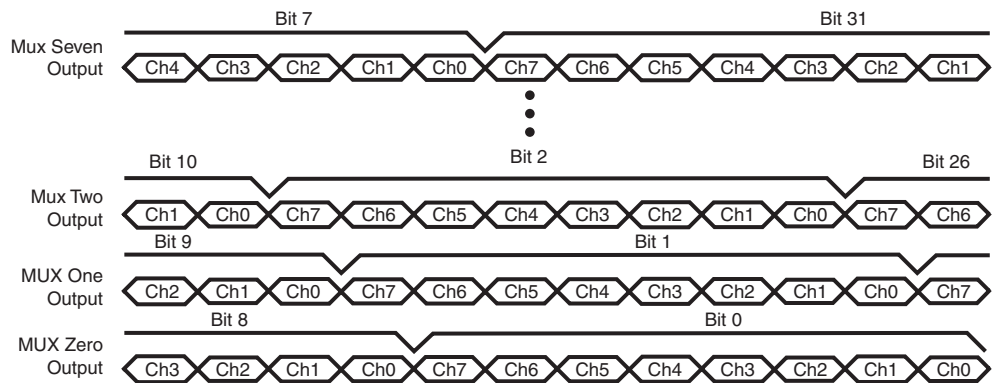
The outputs from the delay lines feed into the multiplexor stage. Each mux is responsible for steering all the “Bit X” bits to the “Bit X” RAMs. As determined from the delay diagram in Figure 3, there is a bit zero available to the bit zero RAM for eight clock cycles, and then a dead space of 24 clocks. Thus, the mux is available to steer data to other banks of RAM during this

“dead period.” Therefore, “Mux Zero” is used to output bits 0, 8 16, and 24; “Mux One” is used to output bits 1, 9, 17, and 25, and so on. See Figure 4 and Figure 5.



x194\_04\_011701

Figure 4: Multiplexor Connections to RAM

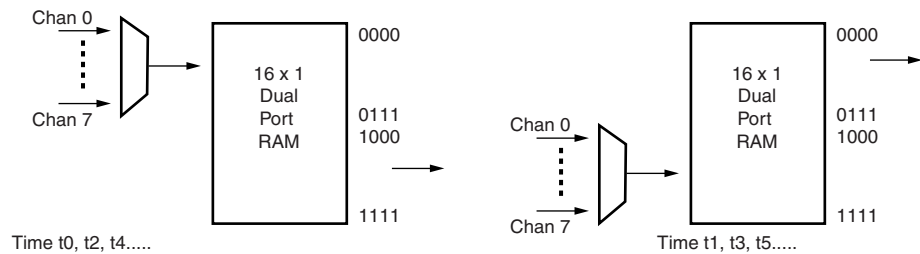


X194\_05\_110100

Figure 5: Multiplexor Output Stream

### Double Buffering for “Free”

The dual port RAM is used to both convert the data from serial-to-parallel format, as well as to provide double buffering. Each of the 16-by-1 dual ports contain all the “Bit X” bits from each channel. The data is written into one-half of the memory and read out from the other half of the memory. The double buffering is accomplished by “ping-ponging” between the memory halves, by control of the read/write address most significant bit (MSB). See Figure 6.



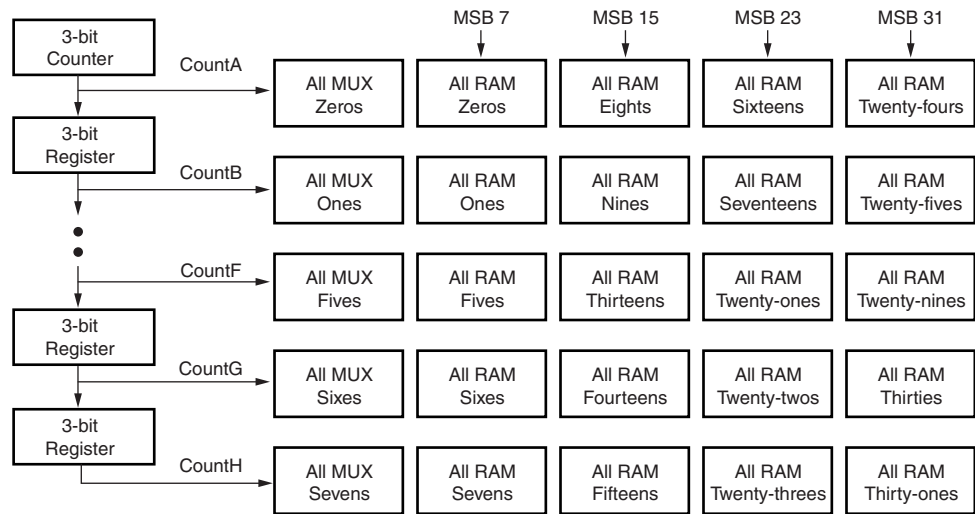
X194\_06\_110100

Figure 6: Ping-Pong Diagram

### Write Control Circuit – Address Generation

The write control circuit consists of an address generation circuit, a write enable generation circuit, and write address MSB control. The control of the lower three bits of the dual port’s write

address, as well as the select lines for the muxes can be generated by a 3-bit counter, followed by seven 3-bit registers, as shown in Figure 7.

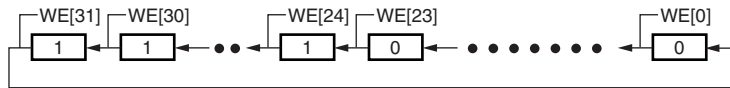


X194\_07\_110100

Figure 7: Write Address Circuit

### Write Control Circuit – Write Enable Generation

The write enables can be generated by using a re-circulating 32-bit shift register, see Figure 8. The shift register is initialized with eight bits preset, and the remainder of the bits cleared. Write enable bit 0 is tied to all “RAM Zeros”; bit 1 is tied to all “RAM Ones”, and so on.

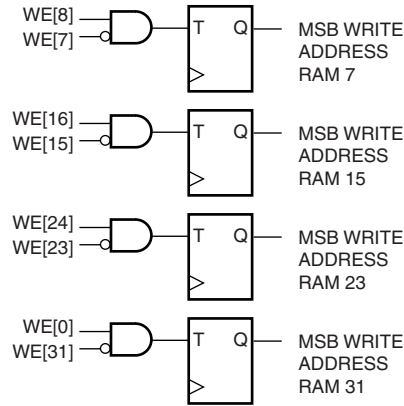


X194\_08\_110100

Figure 8: Write Enable Circuit

### Write Control Circuit – MSB Address Generation

The MSB of the write address is used to control the “ping-ponging” or the dual ports. For the 32-bit example, only four toggle flip-flops and four LUTs are required. This is accomplished by controlling the MSB of RAM 0 through RAM 7 by the MSB control line for RAM 7, the MSB of RAM 8 through RAM 15 by the MSB control line for RAM 15, and so on. The flip-flop toggles after the eight writes to the last bit in the bank are completed. This condition is easily detected, as is shown in Figure 9.



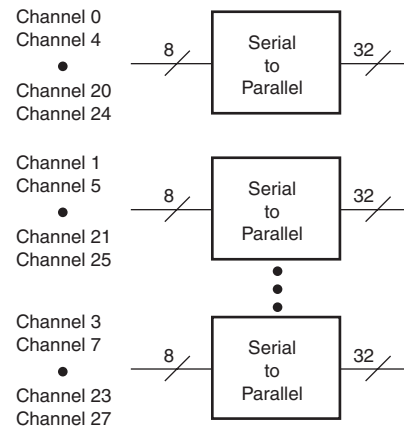
X194\_09\_110100

Figure 9: Write MSB Logic

### Channel Scattering to Reduce Latency

Assuming 32 input channels, scattering the channels across the converter modules, allows for earlier access to the converted data. It takes forty clock cycles to completely convert the data. Conversely, data is available from the first port on each module after thirty-two clocks, and data is available from the second port on each module after thirty-three clocks, and so on. Thus, by scattering the channels, as shown in Figure 10, and by adding a little more sequencing logic in the read circuit, data can be accessed starting at clock 32.

Figure 10

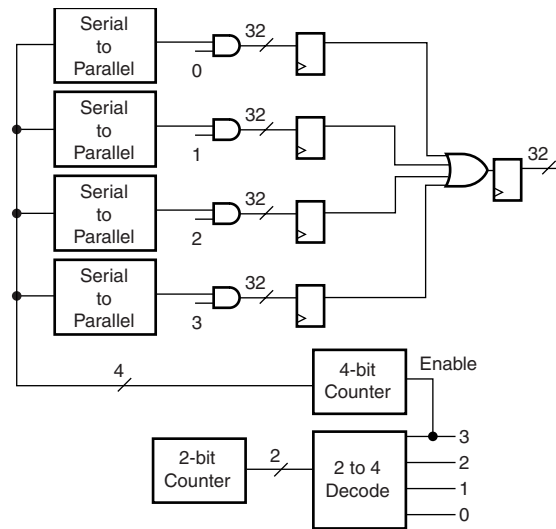


X194\_10\_011701

Figure 10: Channel Scattering

### Read Control Logic

Again, assuming a 32-channel serial to 32-bit parallel conversion, there are four converter modules whose 32-bit output buses need to be multiplexed down to a signal 32-bit bus. This can be accomplished using a 3-state bus structure or by using a gating function. In larger designs, over 150-input channels, a gated implementation has been used, with pipelining to keep read speed at over 150 MHz. At a minimum, the output circuit requires a 2-bit counter, and a 4-bit counter, and the pipeline. The 2-bit counter sequences through the banks, whereas the 4-bit counter sequences through the RAMS, as shown in Figure 11. The 2-bit counter increments every read clock, walking us from bank-to-bank. The 4-bit counter increments when the 2-bit counter reaches its terminal count.



X194\_11\_110100

Figure 11: Read Control

**Design Code**

The reference design code (VHDL and Verilog) is located at [xapp194.zip](#).

**Design Considerations**

The above design has been implemented into a 176-channel design, with the 32-bit output bus running at over 150 MHz. The design required 1,230 flip-flops, and 1,856 LUTS, and fit easily into a Virtex 200 E. A more traditional register-based design would have used over 11,000 flip-flops. There are additional control items to be aware of. Of primary concern is the difference in data rates of the input vs. the output. In the 176-channel design, I was clocking in 176 bits per clock, but only clocking 32 bits of data out at a time. The output clock was not 176/32 faster than the input clock and the input stream had to be stalled to prevent overrun.

Similarly, if the output rate is faster than the input rate, it also must perform handshaking to prevent starvation. The flow control turns out to be fairly trivial, requiring a 2-bit counter, and to signal when the device is empty, half-full, or full. The 2-bit counter is incremented whenever 32 writes have occurred, signified by the WE[31] going high. The counter is decremented whenever the read address reaches its maximum value (both the 4-bit address generator, and the N-bit bank select counter). Thus, the value of the 2-bit counter can be used to provide FIFO-like handshake signals to the front and back ends.

**Revision History**

The following table shows the revision history for this document.

Date	Version	Revision
07/20/04	1.0	Initial Xilinx release.