Chapter 13
Week 7 Class 1

13.1 Serial-Parallel Conversion

We put aside our serdes project a while ago to strengthen our understanding of verilog; now we return to it to implement another part of the deserializer.

13.1.1 Simple Serial-Parallel Converter

We have implemented the serialization frame encoder (Lab 6, Step 8; Week 2 Class 2), and we studied the processing, which is to say, the decoding, of the incoming serial stream when we developed a way of synchronizing a PLL with our framed packet protocol; this was toward the end of Lab 10 (Week 4 Class 1).

So, in terms of our project, we have only a little more to do beyond putting together some things we have done already, and modifying a couple of modules for synthesis. However, let’s look some more into the parallelizing of a serial stream at this point.

Fig. 13.1 Generic serial-parallel converter

The general functionality is given in Fig. 13.1. A parallel bus of output latches or flip-flops should be present, as well as a clock or other synchronizer, and at least one serial input. A purely combinational deserializer is possible, but it would be difficult to use for anything without output synchronization.

There has to be a serial clock, but this may be the same as the parallel clock. The serial clock, if not the same as the parallel clock, may be generated either from the serial side directly, or, as in our serdes project, derived from the serial data.
There should be an internal register to hold partially deserialized data, but this shouldn’t be the same as the one latching the parallel output, unless the design is to allow the fully parallelized data to be available on the output for a duration of less than one clock.

Optional functionality might include a flag announcing when parallel data on the output bus are valid; however, this functionality in principle could be achieved by clock-counting. There should be a `SerValid` input from the serial side to flag when serial data are available to convert. This `SerValid` flag might toggle with each serial bit or byte received; or, instead of such a flag, or we might provide a serial clock based on the assumption that the serial stream can be synchronized with it. If we want to be able to start up the device with a zeroed shift register, we may provide an optional parallel-side reset to do this. We shall assume that the converter could have no control over the serial line’s transmitter, so it would not be meaningful to provide a serial reset.

### Summary of Serial-parallel Conversion Functionality

- Latched parallel data out.
- Parallel-side clock in.
- Serial data in.
- Serial-side clock in (optional).
- Internal deserialization register.
- Parallel-valid output flag (optional).
- Serial-valid flag (optional).
- Parallel-side reset (optional).
- No serial-side reset.

In our serdes project, the serial clock is embedded in the serial stream and must be decoded by the receiver without any `SerValid` flag. This has the obvious disadvantage of requiring some startup overhead before the clock phase and frequency can be established; it has the more-than-redeeming advantage of not requiring any separate clock, with attendant differential phase-lag, separate cross-talk, or related noise issues. So long as the data are usable, the embedded clock will be usable, too. This permits a data stream independent of any other signal in the design; in practice, this is an important factor permitting GHz frequency-range data transmission almost with a zero error rate.

### 13.1.2 Deserialization by Function and Task

If we don’t worry about frame boundaries, parallelization is almost trivial in verilog: We shift in the serial data until we have enough for a parallel word; we unload the shift register onto the parallel bus; and, we continue shifting. We may assume that parallel clocking is fast enough that no serial data will be lost.