Rules for Ready/Valid Handshakes

Ready/valid handshakes are a flexible and lightweight way to connect and control modules in composable ways, but as I designed more complex modules, I found some corner cases I couldn't quite fit into the ready/valid handshake model, and some designs started not composing well because I was implementing the handshakes inconsistently. So, I worked out a set of rules.

I based these rules on those in the AMBA AXI4 Specification, Chapter A3, "Single Interface Requirements", and expanded on their meaning and consequences. I have replaced the inappropriate "master/slave" terminology with "source/destination", correspondingly, when refering to handshake interfaces.

Interfaces

There are two complementary interface types: *source* and *destination*. Both interfaces have three signals: valid, ready, and data. Ready and valid are single-bit signals, and the data signal is of arbitrary width. **All signals are synchronous to the rising edge of the clock**.

- The *source* interface outputs valid and data, and takes ready as input.
- The *destination* interface outputs ready, and receives valid and data.

A connection always goes from the source interface to the destination interface. Other pairings cannot work.

Loops

There must be no combinational paths from input to output signals in the source interface (ready to valid), nor in the destination interface (valid to ready). Otherwise, combinational loops will form when connecting interfaces.

Even if only one type of interface has a loop, thus avoiding combinational loops, the remaining combinational path will go from one interface to the other and back to the first, which will cause a long delay. This delay will both limit your design clock frequency and make placement and routing of the modules connected by these interfaces more difficult. *It is not worth saving a cycle of latency here at the expense of the performance of the rest of the design.* Proper handshake interface design and pipelining will avoid this extra cycle of latency, while *improving* clock frequency and P&R.

Handshake Procedure

At any time, the source interface raises *and holds steady* valid when data is available, and the destination interface raises ready only when it can accept more data. When both valid and ready are high, the handshake is complete, the destination interface accepts the data in the same cycle, and the ready and valid outputs change state if necessary.

The destination interface can freely assert and deassert ready at any time. However, it is beneficial to have the destination interface assert ready as soon as it can accept data, before the source interface asserts valid, to shorten handshakes to a single cycle. For the same reason, the source interface should assert and hold steady valid as soon as it has data to send.

After a handshake completes, the source interface drops valid if it has no more data, otherwise it keeps valid high for the next handshake, which may complete in the same clock cycle. Similarly, the destination interface drops ready if it cannot accept more data, otherwise it keeps ready high and can complete the next handshake in the same clock cycle if valid is also high.

Incorrect Handshakes

It is possible, although incorrect, to have the destination interface accept data at the moment the source interface asserts valid, then signal ready to complete the handshake, after the data is processed, to move the source interface to the next data item.

Although this is legal and will work, it obscures the operation of the logic, and breaks pipelining since the source interface cannot begin processing and presenting the next data item concurrently with the destination interface processing the current data item. If the source and destination processing times are equal, which is the optimum when pipelining, this incorrect handshake will halve the throughput instead of doubling it!

Sampling Changing Data

Typically, the source interface holds data steady alongside valid, changing data only after the handshake completes. Any data value outside of the clock cycle when the handshake completes is lost. It is allowable to have data be a continuously changing value, synchronously to the clock, which will get sampled whenever a handshake completes.

Sampling changing data is best done by having the source interface hold valid high, while the destination interface asserts ready at an interval. The opposite method of sampling where the source interface periodically asserts valid does not guarantee a predictable sampling interval, since it depends on the destination interface having already asserted ready.

Avoiding Deadlocks and Livelocks

We must constrain the behavior of the valid and ready signals to prevent deadlocks, where the source and destination interfaces wait forever for each other to respond, and to prevent livelocks, where both interfaces are responding but the handshake never completes.

To prevent deadlocks, the source interface must not wait until the destination interface asserts ready before asserting valid, while the destination interface can wait for the source interface to assert valid before asserting ready. This waiting will extend the handshake to two cycles, completing in the second cycle, but has applications where we want to selectively complete handshakes from multiple source interfaces (e.g.: an arbiter).

To prevent livelocks, when the source interface asserts valid *it must remain asserted until the handshake completes*, else we could end up in a situation where the source interface temporarily asserts valid and the destination interface temporarily asserts ready, but they never coincide to complete the handshake.

Reset

The reset signal to modules with source and/or destination interfaces can be active high or low, may assert asynchronously, but must deassert synchronously. However, using an active high reset limits the amount of inverted logic to read, which keeps the logic clearer. *Also, I strongly recommend using a fully synchronous reset since an asynchronous reset will inhibit register retiming.* Any latch holding state inside the source or destination interface must be reset, else the interface may remain in the wrong state after reset (e.g., the source interface signaling that data is valid when there isn't any).