Size Control Counter

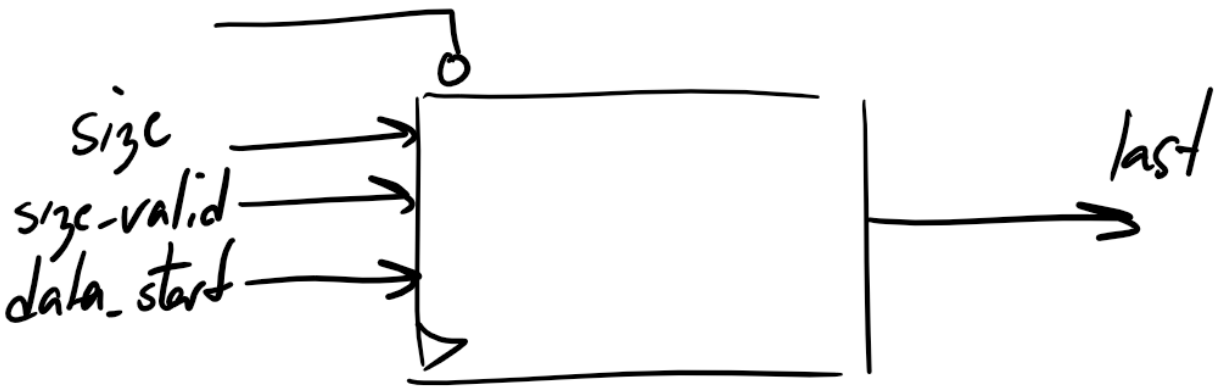
Abstract

This is a short discussion of the design of the size control counter. Two things. First, I've chosen a design that loads the size value into a counter and then counts down, reaching a value where we can know that we're on the last datum. You may choose a different implementation; for example, you could load the value of the size into a register and then use a separate counter to start from a base number and count up until you reach the size.

Second, I chose the verbiage above specifically because the value to count down to or the value to start counting up from may not be zero. This is dependent on an analysis of the timing in the design.

Top Level Block Diagram

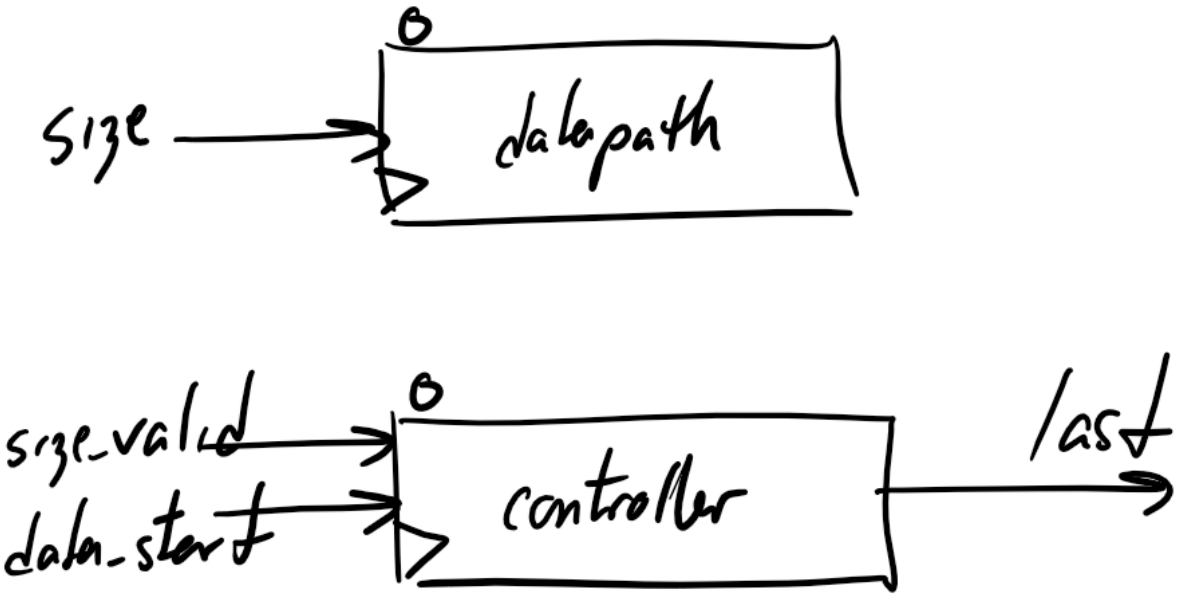
As always, a top level block diagram.



Note that the reset and clock have been left out only to avoid cluttering the diagram.

Second Level Block Diagram

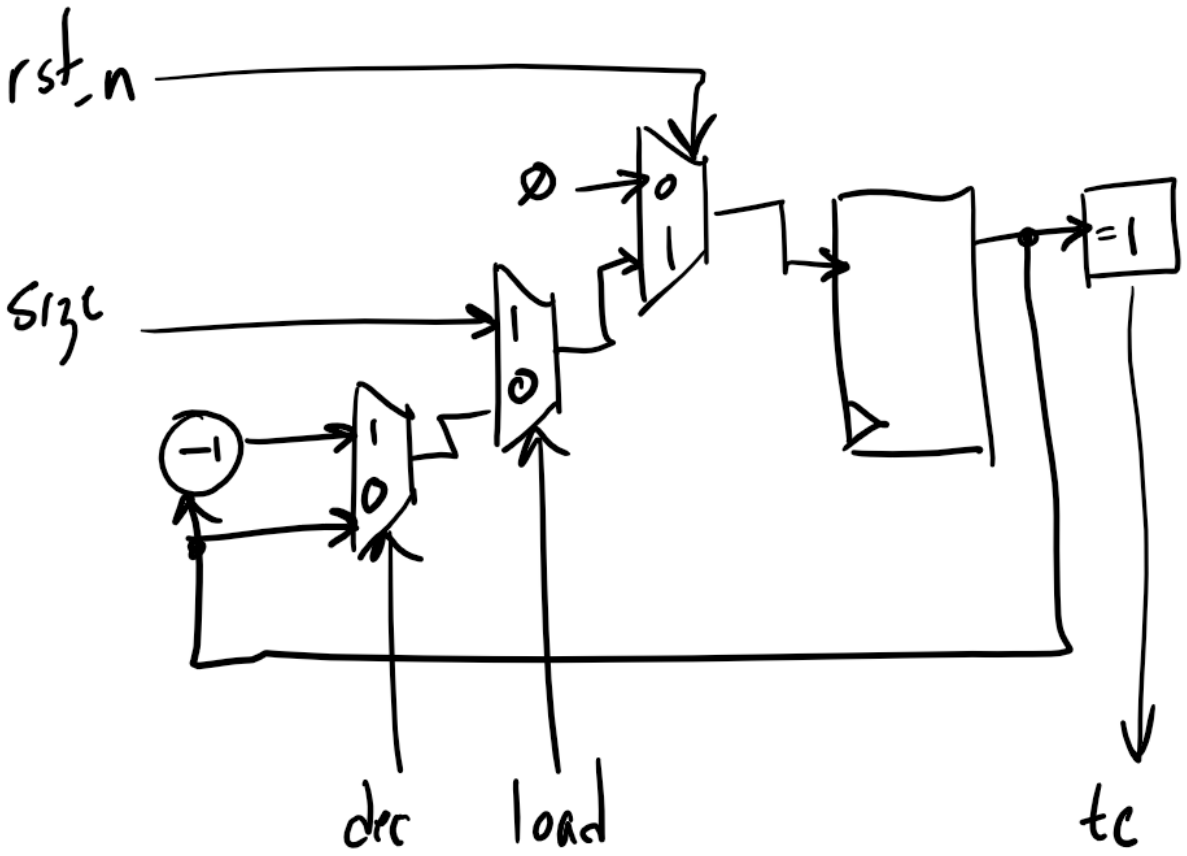
Since there are steps in the process a controller is implied. And, since there is a manipulation of data a datapath is implied. The second level block diagram looks like the following.



Again, I've left out the clock and reset. Notice, too, that there are no signals between the datapath and the controller. We just haven't gotten to that point in the design. We'll add those later.

Datapath

This is the easier of the two for this design. It's a loadable, decrementing counter. There is always a reset. Since there are two choices to be made (plus the reset) it makes sense to think a little about priority. Since there is a load it should probably have higher priority than the decrement. Think of it this way; if I asserted that I wanted to load a value would I also want to be forced to ensure that my decrement control had a specific value? Probably not. Since we won't want to also force decrement to a specific value when loading, load has a higher priority and its mux goes closer to the register.



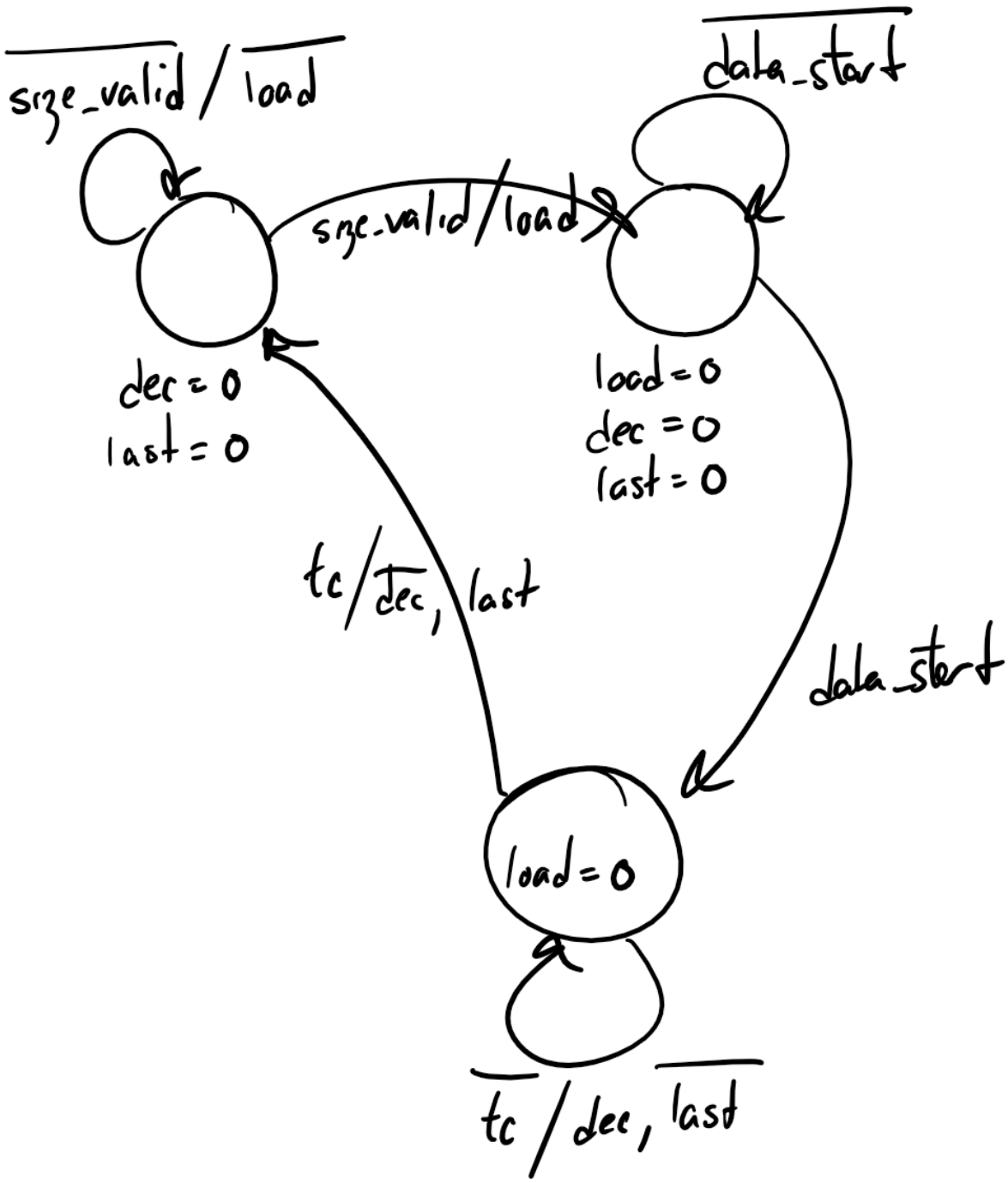
Take a look at the diagram above for the datapath. There is nothing really special about the design. It's just a decrementing counter that you can load with the size. We've added a couple of signals that aren't at the top level.

- **dec**: This is the control that chooses between retaining the same value and decrementing by 1.
- **load**: This higher priority than dec and when asserted will pass the value of the size input towards the register.
- **tc**: This is a terminal count. It provides indication when the counter has reached some boundary.

Notice that the terminal count, **tc**, signal asserts when the value of the counter has reached 1. This is instead of zero. The reason is that we need last to assert over the clock edge in the last datum will accumulate. It's necessary to support being able to drive valid at the correct time in the final project.

Controller

The controller is not particularly complex. There is just a number of things to look out for. Specifically, there is a mix of Mealy and Moore based outputs depending on the state that you're in. For example, while in the first state waiting for **size_valid** to assert you keep load low; then, when **size_valid** asserts you drive load high to be able to store the **size** in that clock cycle. Load is a Mealy output in that first state.



In that first state we are neither asserting last nor decrementing since we're waiting on **size_valid** to even start the rest of the process. In the second state all of the outputs are low because we're just waiting on **data_start** to tell us that the first clock edge to count (first data to accumulate in the project) to be coming in the next cycle. Notice the difference between **size** requiring capture in the same clock cycle as **size_valid** while **data_start** tells us that the first data are coming in the next cycle.

In the third state is where we decrement the counter while we wait for the data path to tell that we're counting the last clock edge. Here, load is a Moore type output because it's always zero. However, both decrement and last are Mealy outputs tied to the value of **tc**. When **tc** is low then we're not done counting down and should continue to decrement while not asserting last. But, when **tc** asserts from the datapath then we know that the clock edge for this cycle will be the last one to count. We use that to stop decrementing, assert last, and move back to the first state to wait for the next **size**.

Implementation

I continue to recommend that you use the 2 procedural block method for implementing state machines. Build a state vector and then a second procedural block to describe the behavior for both the next state and output logic.

What about …?

Of course, this is just one way to implement the design. Some immediate things that I notice that could be changed are

1. You could make load a Moore output in the first state and always store the size until you move to the next state.
2. You could add another state so that there are two states for decrementing. This would allow you to make both decrement and load Moore outputs in both of those states. In the first decrement would be high and load would be low and you would look for tc. The tc signal would have to be changed to compare to 2, though. Then you would move to a final state where decrement would be low and last would be high and you would always move to the starting state.
    a. This is nice because it makes the outputs Moore outputs. But, it suffers from an additional state and you would need another check on the loaded size to see if it was 1. In that case you'd have to skip to the final state on data_start.

There are also some boundary conditions that may not work.

1. What happens if the value of size is zero? This is a thought exercise; I won't send a size of zero in the project.