Exercise 3: Interfaces & Latency

Overview

In this exercise, we will be building two designs. The first will be a multi-cycle implementation of a multiplier. The second will be building a RAM that can support reading and writing the same address in the same cycle. We'll be working with latency insensitive interfaces, which is a commonly used design pattern in more complex hardware designs.

Background

Interface Latency

When you build an interface for a hardware module, you not only have an expectation of what data is provided by each port, but also an expectation of when the values on the port are valid.

Consider the multiplier we'll be implementing and let's imagine it's a 32-bit multiplier. Doing the multiplication in a single cycle is impractical from a hardware perspective (consider what the combinational circuit for a 32-bit multiplication might look like). So we want to be able to take multiple cycles to do the multiplication. So when does another piece of logic using the module know when the output is valid?

One way could be to always wait a certain number of cycles. How many cycles though? Well, if we process the multiplier value one bit at a time, adding the multiplicand to a total depending on whether the bit is 1 or a 0, it could take 32 cycles. But do we always need to take 32 cycles? Consider if we're multiplying by 4 versus 2^31. And what if we want to switch to a different implementation where we process the multiplier value multiple bits at the time? That could also change the timing of the output, so that's not very helpful.

While we're at it, let's consider how the multiplier knows to start a multiplication. Should the multiplier just always use the values on its input after it finishes the multiplication? What if the values on its input change during the multiplication? Should it start over? Or should it finish the multiplication its doing. And furthermore, now the module responsible for providing the values as input to the multiplier might need to track how many cycles the multiplier takes to finish.

This is only one module too. Imagine where we have to construct a whole system where modules may have different interface latencies. This would get pretty out of hand pretty quickly.

Latency-Insensitive Interfaces

Enter: latency insensitive interfaces. In this design pattern, modules provide an interface that provides a simple standardized handshake. Values are exchanged at the interface only in the cycle the handshake

occurs. This allows us easily understand when a consumer is able to accept new values and when a producer is providing valid values for computation.

The typical form of this interface uses a val-rdy (valid-ready) handshake. This means that modules have (at least) the valid and ready control signals associated with their interface. The producer of some values will set the valid signal and the consumer will set the ready signal. If both signals are high in the same cycle, then the exchange of values is complete, and the consumer has taken the values. Now we no longer have to remember how many cycles a module may take to produce a value, for example, because it will just set the valid signal when it is finished.

Let's consider the handshake in the context of providing values to the multiplier. The logic providing the values will set its valid signal high when it has values that it wants to request the multiplier will operate on. In this case, the logic is the producer and the multiplier is the consumer. If the multiplier is ready to operate on a new set of values, it will its ready signal high. If both the valid and ready signals are high in the same cycle, then the multiplier will start to operate on those values.

Since the multiplier also produces an output, it will have a val-rdy interface in which it is the producer and some other logic is the consumer. So when it has calculated the product, it can set its output valid signal high. Once the logic consuming the value sets its ready signal high, the multiplier knows the product has been consumed, and it can start another transaction.

Synchronous vs Asynchronous Memories

While we're talking about latency and interfaces, let's touch on memories, specifically SRAM or register files or or other similar things. Not DRAM. DRAM is a whole other thing. Memory primitives typically don't provide a latency insensitive interface, but they come in two main variants: asynchronous and synchronous. These variants refer to the latency of a read on the memory. That is, when you make a read request, when is the value on the output the value you requested.

For **asynchronous memories**, the value is available in the same cycle you requested in. This is the behavior of a register file. For **synchronous memories**, the value is available in the next cycle after you requested it. You can imagine a register file that has an extra flop on either the input or the output.

Part 1: the multiplier

The task is to implement a multi-cycle multiplier with a parameterizable operand width. It should have a val-rdy interface.

Files

- `multiplier_top.sv`: instantiates the multiplier and is the top-level for simulation
- `multiplier.sv`: multiplier top-level. Look here to see the val-rdy interface
- `multiplier_ctrl.sv`: empty file to implement the multiplier control logic
- `multiplier_data.sv`: empty file to implement the multiplier datapath logic

Implementation

As hinted at above, there are multiple ways to do this, including optimizations. We don't need to get fancy with the actual multiplier logic here. The point is to provide a simple example of a latency-sensitive interface. However, your implementation shouldn't always take the same number of cycles to output the product.

One way to do this could be to start with iterating over the multiplier one bit at a time and adding the multiplicand to a running sum and always iterating over the whole multiplier. The next step could be to terminate when you don't need to iterate over the multiplier any more and provide the product as soon as it's ready.

Testing

Feel free to add more tests. At the moment, the success output looks like:

```
-- RUN ---------------------
obj_dir/Vmultiplier_top +trace
[1] Tracing to logs/vlt_dump.vcd...

[1] Model running...

[20] req_val: 0 A * B = 0 * 0 rd_resp_val: 0, product: 0
Run some basic test cases
Run exhaustive testing
Testing changing inputs while a request is in progress
Test backpressuring the input
```

Note: The test will pause for a bit after printing "Run exhaustive testing", because it just takes a while to run all the different combinations.

Part 2: the memory

The task here is to take the provided synchronous 1r1w memory primitive and build another synchronous 1r1w memory but with some extra features. A 1r1w memory means that there is one port for issuing write requests and another for issuing read requests, so it can accept both a read and write request in the same cycle. The exception is when there is a read and write request for the same address in the same cycle. In this case, the primitive throws an error.

There are two features to add. The first is that we want to provide a val-rdy interface. The second is that we want to be able to handle reads and writes to the same address in the same cycle by bypassing the written value.

Files

- `mem_wr_bypass_top.sv`: top-level for simulation, instantiates the memory
- `mem_1r1w_sync_wr_bypass.sv`: memory top-level. It should contain your logic to implement the val-rdy interface as well as the instantiation of the provided memory primitive

- `mem_1r1w_sync.sv`: provided memory primitive. Do not edit this file.

Implementation

Like the multiplier, the logic here shouldn't be very involved and is more concerned with your understanding of how to implement and use a val-rdy interface.

Start with implementing the val-rdy interface. There is both a val-rdy handshake on the request side and response side. In particular, this means that the response may not be consumed the cycle after, and your logic must make sure somehow that the response remains valid, even if there is another valid request offered on the request interface.

After you've gotten that working, then move onto implementing the bypassing. Consider what you need to save in order to bypass the correct data from write to read as well as how to tell whether you should use the data you've saved or the data from the memory.

Your implemented memory should have the same latency as the original memory primitive in cases without collision and backpressuring. This means that it should be able to accept a write and/or read in every cycle and reads should output in the cycle after they are requested.

Testing

Feel free to add more tests in `sim_main.cpp` as you see fit. At thet moment, the success output looks like

```
-- RUN ---------------------
obj_dir/Vmem_wr_bypass_top +trace
[1] Tracing to logs/vlt_dump.vcd...

[1] Model running...

[20] wr_req_val: 0 mem[0] <- 0 rd_resp_val: 0, rd_resp_data: 3b
[30] wr_req_val: 1 mem[0] <- ab rd_resp_val: 0, rd_resp_data: 3b
[55] wr_req_val: 0 mem[0] <- ab rd_resp_val: 1, rd_resp_data: ab
[70] wr_req_val: 1 mem[1] <- c6 rd_resp_val: 0, rd_resp_data: ab
[95] wr_req_val: 0 mem[1] <- c6 rd_resp_val: 1, rd_resp_data: c6
[110] wr_req_val: 1 mem[2] <- 69 rd_resp_val: 0, rd_resp_data: c6
[135] wr_req_val: 0 mem[2] <- 69 rd_resp_val: 1, rd_resp_data: 69
[150] wr_req_val: 1 mem[3] <- 73 rd_resp_val: 0, rd_resp_data: 69
[175] wr_req_val: 0 mem[3] <- 73 rd_resp_val: 1, rd_resp_data: 73
[190] wr_req_val: 1 mem[4] <- 51 rd_resp_val: 0, rd_resp_data: 73
[215] wr_req_val: 0 mem[4] <- 51 rd_resp_val: 1, rd_resp_data: 51
[230] wr_req_val: 1 mem[5] <- ff rd_resp_val: 0, rd_resp_data: 51
[255] wr_req_val: 0 mem[5] <- ff rd_resp_val: 1, rd_resp_data: ff
[270] wr_req_val: 1 mem[6] <- 4a rd_resp_val: 0, rd_resp_data: ff
[295] wr_req_val: 0 mem[6] <- 4a rd_resp_val: 1, rd_resp_data: 4a
[310] wr_req_val: 1 mem[7] <- ec rd_resp_val: 0, rd_resp_data: 4a
[335] wr_req_val: 0 mem[7] <- ec rd_resp_val: 1, rd_resp_data: ec
[385] wr_req_val: 0 mem[0] <- 65 rd_resp_val: 1, rd_resp_data: 65
```

```
[425] wr_req_val: 0 mem[0] <- 65 rd_resp_val: 1, rd_resp_data: 4a
```