Part 2: Output FIFO [15 points]

The goal of Part 2 is to build and test the Output FIFO component, whose top-level block diagram is seen here (Figure 2.1).

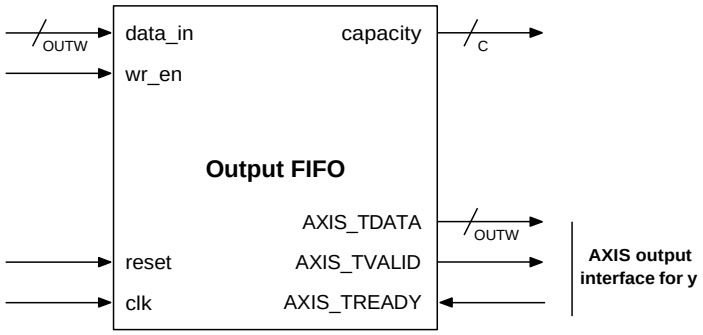

Figure 2.1. Top-level view of Output FIFO module.

This module has two parameters:

- `OUTW`, which represents the number of bits of the FIFO's input and output values. (This will be equal to the `OUTW` of your MAC module.)
  - Your system should support $4 \leq$ `OUTW` ≤ 64.
- `DEPTH`, which determines the number of entries in the FIFO.
  - Your system should support `DEPTH` ≥ 2.

Note in the diagram above, the capacity signal is listed as being `C` bits wide; we will define `C` as:

$$\lceil \log_2(\text{DEPTH} + 1) \rceil$$

or in SystemVerilog code[1]: `$clog2(DEPTH+1)`. If you are surprised by the +1, remember that the capacity can be any number between 0 (the FIFO is full) to DEPTH (meaning the FIFO is empty). So, it needs DEPTH+1 possible values.

In your matrix-vector multiplier, the Output FIFO will be used to hold output vector values computed by the MAC module; they will be stored using the `data_in` port and the corresponding `wr_en` write enable.

The FIFO's output will connect to an AXI-Stream interface with `TDATA`, `TVALID`, and `TREADY`. (Please see Project Overview Section 5 for a specification of this interface.)

You may be wondering why our system needs to use this FIFO at all. Certainly, it would be possible for the output vector values produced by the MAC unit to directly go to the AXI-Stream output interface. The downside to such a design would be that the timing of the computation would then depend heavily on the timing of the external `TREADY` control signal. In other words, if the testbench

[1] Recall, `$clog2()` will compute the $\log_2$ of its operand and round up to the nearest integer. For example, `$clog2(32)=5` and `$clog2(33)=6`.

set the output TREADY to 0, we may have to make our computational module stall, wasting time. (This would also make the control logic needed to control that module more complicated.) Instead, by using a simple FIFO, we can prevent this from happening. As long as there is space in the FIFO, we can compute values and store them there; the FIFO's internal logic can then place available values on the AXI-Stream output whenever the testbench is ready for them.

Recall, we discussed building FIFOs see the slides. This FIFO will behave like that one with one key difference: rather than having an output interface with data_out and rd_en, this FIFO will connect to an AXI stream interface. This means you will be responsible for determining how to adapt the FIFO design from slides to use the TVALID and TREADY control inputs. Some hints:

- The FIFO output is valid if the FIFO is not empty. Use this idea to control the TVALID signal.
- The AXI-Stream interface is reading from the FIFO if TVALID and TREADY are both asserted on the same positive clock edge. Use this idea to determine how to set the rd_en signal.

Your FIFO should be structured like the FIFO we described in slides, with the changes needed to interface its output with AXI-Stream.

Memory

Your FIFO must use one instance of the *dual-port* memory structure we discussed in slides, with synchronous reads and writes and two address ports. Use the following memory module. You can copy this module from:

/home/home4/proj/part2/memory_dual_port.sv

You may not modify this memory module (although you can feel free to copy/paste the code into another .sv file if that's more convenient).

```
module memory_dual_port #(
        parameter                WIDTH=16, SIZE=64,
        localparam               LOGSIZE=$clog2(SIZE)
    )(
        input [WIDTH-1:0]        data_in,
        output logic [WIDTH-1:0] data_out,
        input [LOGSIZE-1:0]      write_addr, read_addr,
        input                    clk, wr_en
    );

    logic [SIZE-1:0][WIDTH-1:0] mem;

    always_ff @(posedge clk) begin
        data_out <= mem[read_addr];
        if (wr_en) begin
            mem[write_addr] <= data_in;
            if (read_addr == write_addr)
                data_out <= data_in;
        end
    end
endmodule
```

There are several important things to understand about this memory:

- The memory is parameterized by two parameters:
    a. `WIDTH`, the number of bits of each word
    b. `SIZE`, the number of words stored in memory

- The memory has a "local parameter" called `LOGSIZE`, which represents the number of address bits needed to address `SIZE` entries. This is automatically computed as the $\log_2$ of `SIZE`, rounded up.

- All reads and writes are synchronous (that is, they will occur on a positive clock edge).

- The memory has one read port and one write port, and each uses a separate address input. This means the memory can read and write from two independent locations at the same time.

- The memory has "bypass logic" that means if you are reading and writing to the same address at the same time, you will get the *new* data, not the old data. This is helpful in FIFOs (as discussed in Topic 8).

Remember, you can overwrite these parameters when you instantiate the module. For example, if you instantiate the memory as:

```
memory_dual_port #(12, 256) myMemInst(clk, din, dout, wr_addr,
      rd_addr, wren);
```

Then you would be building a memory with 256 words, each with 12 bits.

Figure 2.2 demonstrates the timing of reading from the memory. On each positive clock edge, the system samples the value on `rd_ddr`. A short time after the clock edge, the memory will output the value in memory at that location. In this diagram, `mem[7]` represents the value stored in address 7 of the memory.

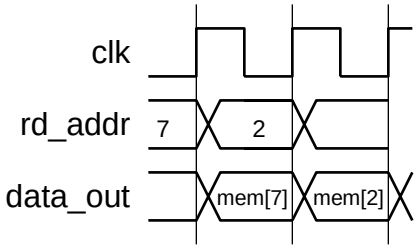

Figure 2.2. Timing of memory read.

Figure 2.3 demonstrates the timing of writing to memory. In this example, you are first writing the value 3 to address 6. Then, on the following cycle, no write is performed because the `wr_en` signal is 0 on the clock edge. Then, value 4 is written to address 1 on the third positive clock edge.

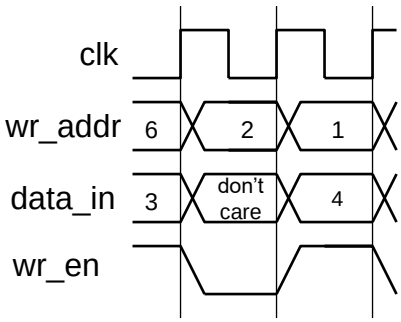

Figure 2.3. Timing of memory write.

Recall from class that this RTL memory module will not synthesize to SRAM—instead it will produce a structure built from of flip-flops. If these memories were large, this would be very inefficient. However, the memories you will need in this project will be fairly small, so we will simply let the logic synthesis tool implement them using registers.

Code

Store all of your files for part2 in a subdirectory called `part2/` . Implement this design in a file called `output_fifo.sv` and use the following top-level module name, port names, and port declarations:

```
module output_fifo #(
        parameter OUTW=32,
        parameter DEPTH=33,
        localparam LOGDEPTH=$clog2(DEPTH)
    )(
        input clk, reset,
        input [OUTW-1:0] data_in,
        input wr_en,
        output logic [$clog2(DEPTH+1)-1:0] capacity,
        output logic [OUTW-1:0] AXIS_TDATA,
        output logic AXIS_TVALID,
        input AXIS_TREADY
    );
```

Testbench

You are provided with a testbench to test your `output_fifo` module. This testbench will randomly write data into the FIFO and read data from it, checking the result. You can find the testbench at

```
/home/home4/proj/part2/output_fifo_tb.sv
```

Copy this file into your `part2/` work directory where your part2 designs are. Please read the testbench code and its comments.

The testbench module includes five parameters. The first three are straightforward:

- `OUTW`: the number of bits for each data word
- `DEPTH`: the number of entries in the FIFO
- `TESTS`: the number of inputs to simulate

The other two parameters are slightly different, because they control the random behavior of the testbench.

- `WRITE_EN_PROB` is a decimal value that represents the probability that the testbench will attempt to write a value into the FIFO on any clock cycle. (If the FIFO is full, the testbench will never write data.)

- `TREADY_PROB` is a decimal value that represents the probability that the testbench will assert `AXIS_TREADY` on any clock cycle.

These parameters should be between 0.001 and 1, inclusive. For example, if you set `WRITE_EN_PROB` to 0.3, then there will be a 30% chance that the testbench will try to write a value into the FIFO on each cycle. If you set either of these parameters to 0, the testbench will randomly pick a value at the beginning of the simulation (and tell you what it chose).

Adjusting the probabilities of these signals is important, and it's especially important to test scenarios where the is a mismatch between them. For example, make sure you test a situation where `WRITE_EN_PROB` is small like 0.01 and `TREADY_PROB` is large like 0.99. This will test the situation where the FIFO is almost always empty—as soon as you write data to the FIFO, the testbench will read it. Similarly, if you reverse the parameters, then you will simulate the situation where the FIFO will quickly fill up with data. Run tests with different probabilities to verify your control logic works correctly. Also be sure you check different values of `DEPTH` and `OUTW`. Importantly, be sure to test the `DEPTH` parameter with numbers that are both powers of 2 and non-powers of 2.

Compile and simulate your code similar to Part 1 (although here there is no C code to compile, since this testbench does not rely on DPI like Part 1 did).

When you are synthesizing your design, keep in mind that this is not real SRAM. Here we are synthesizing a logical description of the memory, but logic synthesis will produce flip-flop based logic with the same logical functionality of the memory. (See Topic 8 slides.)

Report and Code Submission

After implementing and simulating the designs with a variety of parameters, complete the following tasks. In your report, provide the requested information and answer the following questions.

1. The basic form of the FIFO was discussed in slides, but here you needed to adapt that to interface its output with AXI-Stream. Explain how you did that and how your logic works.

2. Synthesize the design with `OUTW`=32 and `DEPTH`=33 using Synopsys DesignCompiler. Find the maximum possible clock frequency. Correct any synthesis problems you find. In your report, give the maximum clock frequency and the area, power, and critical path location for this frequency. Note that the critical path location may be somewhat confusing. Make sure you carefully trace it so you can explain what logic the critical path includes.

   Here you only need to report statistics for the highest clock period. Save your synthesis report with a descriptive name and include it with your submission.