Part 4: Matrix-Sparse Vector Multiplier (MSpVM) [25 points]

The goal of this task is to integrate the components you designed in Parts 1, 2, and 3 and add additional control logic to make a hardware unit that performs MSpVM. Recall, Figures 1 and 2 in Project Overview Section 4 illustrate the top-level ports and a high level block diagram for this top-level system. Your matrix-sparse vector multiplier's top-level module should be named `MSpVM`, and it is parameterized by parameters `M`, `N`, `INW`, and `OUTW`, as defined previously. (For the legal range of these parameters, please see Project Overview Section 4.)

Figure 4.1 illustrates a partial view of how this module is constructed. Here, you can see that the top-level AXI-Stream input interface directly connects to the Input Memory module, and the AXI-Stream output interface directly connects to the output FIFO. The diagram also shows how the sub-module parameters are set. Most are obvious—`INW`, `OUTW`, `M`, and `N` in the sub-modules should match the corresponding parameters in the top-level module. One non-obvious thing to note: your Output FIFO module should have its internal `DEPTH` parameter set to `M`. (This means the FIFO is large enough to fully hold one full output vector.) Also note that the MAC unit is the pipelined `mac_pipe` system you designed in Part 1.2.

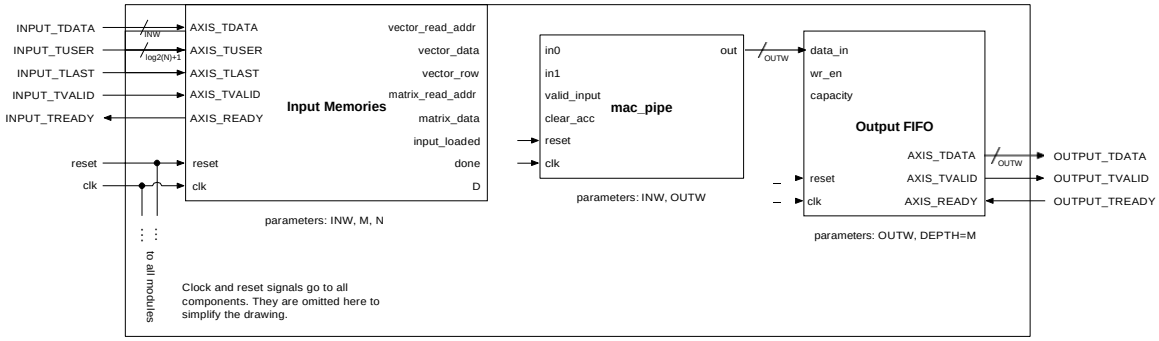

Figure 4.1. This shows a partial view of how your top-level `MSpVM` module should be constructed. You will need to design logic to interact with the signals that are unconnected in this diagram. (You may need to zoom in on the PDF to read this diagram clearly.)

Your job in this task is to design the control logic and interconnections that will allow these three modules to work together to perform matrix-sparse vector multiplication. Your new logic will need to interact with the signals that are unconnected in this diagram.

Recall from Project Overview Section 3 that the MSpVM module must perform the computation seen in this pseudocode:

```
for m = 0 ... M-1:
   y[m] = 0
   for d = 0 ... D-1:
      n = row[d]
      y[m] += W[m][n] * val[d]
```

Now, we can connect our understanding of this pseudocode to the hardware components in our system. The input vector is represented by `row[d]` and `val[d]`, which you can read from the vector memory in the Input Memory module. The matrix is represented by `W[m][n]`, which you can read from the Input Memory module's matrix memory. The computation in the last line is performed by the MAC module, and the final vector value will be written into the output FIFO. Your control logic will be responsible for making this happen.

Module Operation

Here we will give a brief overview of this module's phases of operation and the sequence of steps it will take. You must construct control logic that will make the system undergo the following steps. Your control logic will likely need to include an FSM and counters. The following phases are not necessarily the only FSM states your system will use, but you should use them as a guide for how your control logic for this module should operate.

1. First, the Input Memory module will interact with the input AXI-Stream interface to load the data into the matrix and vector memories. When the input values are loaded and available in those memories, the Input Memory module will assert its `input_loaded` output signal.

2. Then, your control logic will be responsible for reading the matrix and sparse vector data from the input memories the `matrix_read_addr` and `vector_read_addr` signals and feeding the correct data into the MAC module. Refer to the pseudocode above to think about how the matrix addressing should work.

   Also recall that the matrix will be stored in row-major order, so you will need to map `W[m][n]` to an address. This simply means that `W[0][0]` is address 0, `W[0][1]` is address 1, `W[0][N-1]` is address N-1, `W[1][0]` is address N, and so on.

3. As your system feeds the matrix and vector values into the MAC unit, the MAC unit will perform the multiply-accumulate operation. Your logic will need to control the MAC module's control inputs (`clear_acc` and `valid_input`).

4. The MAC unit's output will connect to the Output FIFO's input signal. As the FIFO receives output data, its internal logic will send output data onto the output AXI-Stream interface.

   Your control logic will need to set the value of the FIFO's `wr_en` appropriately, as well as monitor the FIFO's capacity to make sure there is space in the FIFO before computing the value.

   Notice that our specification above shows that the FIFO's depth should be `M`—this means the FIFO is capable of holding an entire output vector. So, an easy way to make sure you don't overfill the FIFO is to wait until the FIFO's reported `capacity` is `M` before you start computing a new MSpVM. (Other approaches are possible; one alternative would be to check that the FIFO isn't full before you finish each individual output vector entry, but this makes the control logic much more complex, so I don't recommend it.)

5. After your system is done computing the MSpVM operation, the control logic should set `done` (the input signal to the memory module) to 1 for one clock cycle. This will cause the

`input_memory` module to start reading in new inputs again, and the whole process can repeat.

Efficient Reading from Memories

One place where your system could lose efficiency if you are not careful is in the sequencing of how you read vector and matrix values from their internal memories. The portion of the pseudocode relevant to this operation is:

```
n = row[d]
y[m] += W[m][n] * val[d]
```

Notice that you cannot read the matrix value `W[m][n]` until you have first `row[d]`. Since our memory reads are synchronous, this means you cannot do both of those operations within the same clock cycle. A naïve, but inefficient way to approach this would be to use one cycle for reading `row[d]` and another cycle for reading `W[m][n]`. That would look like this:

- Cycle 0: read `row[0]` and `val[0]` from vector memory
- Cycle 1: read `W[m][row[0]]` from matrix memory
- Cycle 2: read=`row[1]` and `val[1]` from vector memory
- Cycle 3: read `W[m][row[1]]` from matrix memory

…

- Cycle `2*D-2`: read `row[D-1]` and `val[D-1]` from vector memory
- Cycle `2*D-1`: read `W[m][row[D-1]]` from matrix memory

So, this would take 2**D* cycles to read *D* sets of values and send them to the MAC. This means that half of the time, the MAC unit is sitting idle with no useful work to do.

Instead, you can overlap reading from the sparse-vector memory with reading from the `W` memory, like the following sequence:

- Cycle 0: read `row[0]` and `val[0]` from vector memory
- Cycle 1: read `W[m][row[0]]` while concurrently reading `row[1]` and `val[1]`
- Cycle 2: read `W[m][row[1]]` while concurrently reading `row[2]` and `val[2]`

…

- Cycle `D-1`: read `W[m][row[D-2]]` while concurrently reading `row[D-1]` and `val[D-1]`
- Cycle `D`: read `W[m][row[D-1]]` from matrix memory

This sequence will take *D*+1 cycles to read *D* pairs of values and send them to the MAC. This is obviously much more efficient than the naïve method above, which takes 2**D* cycles.

Make sure that your control logic is constructed to make efficient use of the memories and MAC unit while performing computation.

Code

Store all of your files for part4 in a subdirectory called `part4/` . You will need to copy your Parts 1–3 designs here as well, since Part 4 obviously uses them.

Implement your top-level MSpVM module in a file called `MSpVM.sv`. Feel free to use other files and organize your control logic and other modules in any way that makes sense to you, but make sure everything is commented and named clearly. Make sure all files used in Part 4 are located in the `part4/` directory.

Use the following top-level module name, port names, and port declarations:

```
module MSpVM #(
        parameter INW = 8,
        parameter OUTW = 32,
        parameter M=24,
        parameter N=25,
        localparam LOGN = $clog2(N),
        localparam LOGMN = $clog2(M*N)
    )(
        input clk, reset,

        input [INW-1:0] INPUT_TDATA,
        input           INPUT_TVALID,
        input           INPUT_TLAST,
        input [LOGN:0]  INPUT_TUSER,
        output          INPUT_TREADY,

        output [OUTW-1:0] OUTPUT_TDATA,
        output            OUTPUT_TVALID,
        input             OUTPUT_TREADY
    );
```

Testbench

You are provided with a testbench to test your `MSpVM` module. This testbench will generate random data (matrices and sparse vectors) and the expected output vectors. It will then provide the inputs to your module using the input AXI-Stream interface (with random timing on the `INPUT_TVALID` signal) and receive your module's outputs on the output AXI-Stream interface (with random timing on the `OUTPUT_TREADY` signal). The testbench will check your module's output values against the expected results and report any errors to you.

You can find the testbench in the following two files.

```
/home/home4/proj/part4/MSpVM_tb.sv
/home/home4/proj/part4/test_helper.c
```

Copy these files into your `part4/` work directory where your part 4 design is. Before you use it, please read the following brief explanation of how the testbench works. Then read through the testbench files and read the comments. (Note: this `.c` file is identical to the one used in Part 4.)

The testbench module includes six parameters, which will look familiar after your experience with the previous testbenches:

- `INW`: the number of bits for each input data word
- `OUTW`: the number of bits for each output data word
- `M`: the number of rows in the matrix
- `N`: the number of columns in the matrix
- `TESTS`: the number of inputs to simulate
- `INPUT_TVALID_PROB`: a decimal value that represents the probability that the testbench will assert `INPUT_TVALID` on any given cycle. This should be between 0.001 and 1, inclusive. If you set this parameter to 0, the testbench will randomly pick a value at the beginning of the simulation (and tell you what it chose).
- `OUTPUT_TREADY_PROB`: a decimal value that represents the probability that the testbench will assert `OUTPUT_TREADY` on any given cycle. This should be between 0.001 and 1, inclusive. If you set this parameter to 0, the testbench will randomly pick a value at the beginning of the simulation (and tell you what it chose).

The testbench uses a SystemVerilog class called `testdata`. This class is similar to the class with the same name in the Part 3 testbench, but this also includes extra functions that will generate random input data and the expected output data. If you would like to learn more about how this works, please see the code and comments in the testbench.

For each test, the testbench will randomize the test data and generate the expected corresponding output data. The testbench will feed the test input data into the system via the input AXI-Stream interface and receive the output data on the output AXI-Stream interface. The timing of the input's `TVALID` and the output's `TREADY` will be randomized based on the probability parameters described above.

The testbench will check each output value and report any errors to the screen. The testbench will also report your system's simulated throughput in terms of MSpVMs per cycle. We will discuss this more in the "Throughput" subsection below.

Compile and simulate your code, using a variety of different parameters. Don't forget that you can set top-level testbench parameters from the command like with `-G` like `-G INW=32`. (For more examples, see the description of the Part 1 testbench.) Make sure your system works correctly across the legal range of `M`, `N`, `INW`, and `OUTW` (defined in the Project Overview).

It's also important to adjust the `INPUT_TVALID` and `OUTPUT_TREADY` probabilities. For example, if `INPUT_TVALID` has high probability and `OUTPUT_TREADY` has low probability, your system will behave differently than if the probabilities were reversed. (In the first case, the system will be slow at outputting data, so it will simulate what happens if your FIFO fills up and the system needs to stall. In the second case, you will be checking that your logic works correctly when it must frequently stall due to slow input loading.) Make sure you simulate a variety of different scenarios.

Overflow

Recall from Part 1 that the accumulator in a MAC unit can overflow if `OUTW` is too small to hold the result of the operations performed on it. Here we will deal with this by detecting overflow in the testbench. When it calculates the expected results, it will detect when this happens and print a message to the screen that looks like this:

```
# WARNING: Output overflow. You must increase OUTW or decrease INW for
correct output. Value=52267. Current OUTW=16 --> values must be between
-32768 and 32767.
```

If you see this message, this doesn't mean there is a problem with your design; it just means that it is not possible to compute the MSpVM with the values of `OUTW` and `INW` you provided. In this case, you should make `INW` smaller or `OUTW` larger and try again.

Throughput

We will characterize the speed of your designs based on their throughput. Recall, throughput is a metric that quantifies the rate that a system processes inputs or performs computations. We will measure throughput in terms of MSpVMs per second.

For example, if your system performs an MSpVM in 1000 cycles, and it has a maximum clock period of 1ns, we calculate its throughput as:

$$\frac{1 \text{ op}}{1000 \text{ cycles}} \times \frac{1 \text{ cycle}}{10^{-9} \text{ sec}} = 10^6 \frac{\text{ops}}{\text{sec}}$$

(where one "op" represents one full MSpVM operation).

So, this hypothetical system would compute 10^6 = 1 million MSpVMs per second.

The number of cycles required by your MSpVM systems will depend on several external factors:

- The values of parameters *M* and *N*
- The number of non-zero values *D* in your sparse matrix
- How frequently the testbench asserts `INPUT_TVALID` and `OUTPUT_TREADY`
- Whether the system needs to load a new matrix (`new_matrix==1`) or use the previously stored matrix (`new_matrix==0`)

When we measure throughput, we will choose values for hardware parameters `M`, `N`, `INW`, and `OUTW`. Then we will choose values for testbench parameters `INPUT_TVALID_PROB` and `OUTPUT_TREADY_PROB`. We will measure the cycle count over many operations. `TESTS = 10000` is a good number. Although `D` and `new_matrix` are randomized for each test, over the course of 10000 tests, their average values will be stable: `D` is randomly selected between 1 and `N`, so its average value will be `(N+1)/2`; `new_matrix` is randomly 0 or 1 with equal probability.

So, we will compute the throughput averaged over 10000 test cases, e.g.:

$$\frac{10000 \text{ ops}}{10^7 \text{ cycles}} \times \frac{1 \text{ cycle}}{10^{-9} \text{ sec}} = 10^6 \frac{\text{ops}}{\text{sec}}$$

How will you find the cycle count? Obviously, you aren't going to count millions of cycles. Instead, the testbench we provide will automatically count these cycles and show you the result at the end of the simulation. For example,

```
# Your system computed       10000 MSpVMs in    10193214 cycles
```

So here, you would have 10000 ops in 10,193,214 cycles. Next, you would synthesize this design to find the minimum clock period and compute the number of MSpVMs per second. In the questions below, you will be asked to do this for some specific configurations.

Report and Code Submission

After implementing and simulating your Part 4 design with a variety of parameters, complete the following tasks. In your report, provide the requested information and answer the following questions.

1. This part of the project required you to design a significant amount of control logic that interacts with the existing modules. Carefully and thoroughly document how your top-level system works. Your documentation should allow the reader to fully understand how it works without looking at the code.

2. In Part 3, you wrote equations that described the number of cycles the `input_mems` module requires (given `M`, `N`, and `D`) when `new_matrix==0` and `new_matrix==1`.

   Now, you should use your understanding of your system to update these equations to describe the number of cycles for the entire MSpVM operation in the same scenarios. Your equation should reflect the best-case number of cycles between when your system starts receiving inputs and when it is done with that computation and ready to receive a new set of inputs (where "best case" implies that `INPUT_TVALID` and `OUTPUT_TREADY` are always 1).

   Based on these equations, is your system's performance limited by any one phase of execution? (For example, if your input loading time is much higher than the compute time, this shows that the input loading time limits your speed much more than computation. On the other hand, if the system spends most of its time doing MAC operations on data, then the computational unit is the limiting factor. If the times are close to balanced, then this shows your system's performance is highly dependent on both of them.)

   Justify and explain your answers.

3. Here, you will synthesize your MSpVM design with DesignCompiler. You will synthesize designs with the following sets of parameters:
   - `INW`=12, `OUTW`=36, `M`=7, `N`=9
   - `INW`=24, `OUTW`=64, `M`=17, `N`=15

   For each design, report the maximum clock frequency, minimum clock period, area, and power. For each, describe where the critical path is in the design. (Make sure you explain the critical path fully; don't just list the start and end points.) You only need to report data for the smallest clock period you were able to find for each design. Save these two synthesis reports with descriptive names and include them with your submission.

4. Now, find the throughput of each of the two designs you synthesized in question 3. Evaluate each of the two designs under two different assumptions about testbench parameters `INPUT_TVALID_PROB` and `OUTPUT_TREADY_PROB`: 0.25, 0.5, 0.75, and 1. (That is, do four simulations for each design, one where both `_PROB` parameters are 0.25, one where they are both 0.5, and so on.)

For each, record the number of clock cycles needed for 10,000 MSpVMs, as reported by the testbench with the parameters set appropriately. Then use those cycle counts along with the clock periods you found from synthesis, to find the throughput in number of MSpVMs per second for each design under the four assumptions of the `_PROB` parameters.

In your report, make a table that shows the cycle counts and computed throughputs for all 8 scenarios (two designs with four sets of assumptions each). Make sure your tables include units (here and in all questions). For example, this table could look like the following:

	Design 1 (7x9 matrix)		**Design 2 (17x15 matrix)**	
`TVALID` and `TREADY` prob	cycle count	throughput (ops/sec)	cycle count	throughput (ops/sec)
0.25				
0.50				
0.75				
1.00				

5. The average delay of a system is the average amount of time that elapses between when it starts a computation and when it finishes it. For the 8 scenarios evaluated in question 4, determine the delay in seconds (or ms, μs, ns, etc., as appropriate). Use the cycle counts you determined in the previous question and the clock period you determined in question 3. (Don't forget that the cycle counts reported in question 4 are for 10,000 MSpVMs—you need to find the average delay for a single MSpVM). Report these delays in a table.

6. The synthesis tool gives you an estimate of the power of your system. Use the power obtained from synthesis and the delays you computed in question 5 to determine the average energy your system consumes per MSpVM for each of the eight scenarios you have evaluated in the previous questions. Report these values in a table.

   Remember: energy is measured in joules. Power = energy per time. 1 Watt = 1 Joule * 1 second.

7. A joint metric that combines the effects of area and speed in a single value is the *area-delay product*. The area-delay product is found by multiplying the area of the system times its delay. (Since these are both metrics that we want to minimize, lower area-delay products are better than higher ones.) Calculate the area-delay product of your system under these eight scenarios and report the results in a table.