Project Overview:

Matrix-Sparse Vector Mult. Hardware

This project specification is contained in six documents. This document contains the overview of the project. (You should start here.) Please see other accompanying PDFs for detailed specifications and tasks for each of the five tasks ("Part 1" through "Part 5") of the project.

1. Introduction

In this project, you will design, implement, simulate, and synthesize a hardware system for performing matrix-vector multiplication, where the matrix is dense, and the vector is sparse. This is called "matrix-sparse vector multiplication," which we will abbreviate as **MSpVM**. You will turn in:

- your documented and commented code
- clearly labeled synthesis reports
- a report answering all questions and including requested information

I will run additional simulations on the code you turn in, so it is very important to:

1. Make sure your designs simulate correctly using QuestaSim ***on the lab computers or the CAD servers***.
2. Carefully organize your code as specified in this document.
3. Make sure the names and behavior of all signals match this specification ***exactly.***
4. Carefully label and document your code.

Your project will be evaluated on correctness and efficiency of your design, the quality of your report, and your answers to questions in the report.

You may work alone or with one partner on this project. **You may not share code with others (except your partner). This means you may not allow others to see your code, nor may you read others' code (for this or related projects). All code will be run through an automatic code comparison tool. Plagiarism will result in a score of zero on the assignment for all involved partie**

File Organization

This project is broken into five parts. To make it possible for me to grade and understand your work, please carefully organize your files. Use a separate sub-directory for each part (called `part1/` `part2/` `part3/` `part4/` and `part5/`). Then be sure to name your files and modules as specified in the description below. Make sure all your files are stored inside of a private work directory like the `ese507work` directory you made in the HW2 Tool Tutorial.

Point Breakdown

1. Part 1: Multiply-Accumulate Unit [15 points]
2. Part 2: Output FIFO [15 points]
3. Part 3: Input Memory Module [20 points]
4. Part 4: Matrix-Sparse Vector Multiplier (MSpVM) [25 points]
5. Part 5: Throughput Optimization [20 points]
6. Quality of report, code, comments, and organization [5 points]

Getting Started

As you can see, this project is large and complex. This document provides a high-level overview and some important background information. Then, the accompanying documents give the specification and tasks for each of the five parts of the project. Begin by carefully reading this overview, and then you should spend some time looking through Parts 1–5. Then when you are ready to start working, see the Part 1 document.

3. Background

3.1 Matrix-Vector Multiplication

We first begin by reviewing matrix-vector multiplication. As an example, let W represent a square 3×3 matrix, and let x represent a (column) vector of length 3. The product $y = W x$ is defined as:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} w_{0,0} & w_{0,1} & w_{0,2} \\ w_{1,0} & w_{1,1} & w_{1,2} \\ w_{2,0} & w_{2,1} & w_{2,2} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} w_{0,0}x_0 + w_{0,1}x_1 + w_{0,2}x_2 \\ w_{1,0}x_0 + w_{1,1}x_1 + w_{1,2}x_2 \\ w_{2,0}x_0 + w_{2,1}x_1 + w_{2,2}x_2 \end{bmatrix}$$

So, this system takes in 12 values (the 3×3 matrix W_3 and the 3×1 column vector x) and produces 3 values (3×1 column vector y).

We can also use array notation and represent this operation as computing (for $m = 0,1,2$):

$$y[m] = \sum_{n=0}^{2} W[m][n] \cdot x[n]$$

So, each output value $y[m]$ is computed by multiplying and adding the appropriate values of the matrix W and input vector x.

3.2 Matrix-Vector Multiplication with Generalized Dimensions

In this project, you will consider matrix-vector multiplications parameterized by the matrix and vector dimensions. Specifically, let W be a matrix with M rows and N columns, let x represent a column vector of length N, and let y represent a column vector of length M. Then we can represent this operation as:

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{M-1} \end{bmatrix} = \begin{bmatrix} W_{0,0} & W_{0,1} & \dots & W_{0,N-1} \\ W_{1,0} & W_{1,1} & \dots & W_{1,N-1} \\ \vdots & \vdots & \ddots & \vdots \\ W_{M-1,0} & W_{M-1,1} & \dots & W_{M-1,N-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{bmatrix}$$

Or, in array notation:

$$y[m] = \sum_{n=0}^{N-1} W[m][n] \cdot x[n], \quad \text{for } m = 0, \dots, M-1$$

Or, in pseudocode:

```
for m = 0 ... M-1:
   y[m] = 0
   for n = 0 ... N-1:
      y[m] += W[m][n] * x[n]
```

Computing each of the M values in y requires performing N multiplications and summing up their results. In total, this requires MN multiplications and $M(N-1)$ additions.

In this project, M and N will be parameters of your hardware system. That is, you will design a system in SystemVerilog that has parameters M and N which can be changed in the code.

3.3 Matrix-Sparse Vector Multiplication (MSpVM)

We say a matrix or vector is considered *sparse* if many of its elements are equal to 0. If a matrix/vector is not sparse, we call it *dense*. Sparse data occurs in a very wide number of applications in science and engineering such as solving partial differential equations, circuit simulation, graph theory, and machine learning. Often, these applications operate on very large, very sparse matrices. In this project, you study a simpler (but useful) variation of this problem: multiplication of a relatively small dense matrix with a sparse vector. Specifically, this problem is inspired by recent research on sparsity in transformer networks (e.g., GPT), which are commonly used in natural language applications such as chatbots.

We will use D to denote the number of non-zero entries in vector x. (Necessarily, $1 \leq D \leq N$.) Here is an example of MSpVM of a dense 3x4 matrix with a sparse vector that has two non-zero entries. (That is, M=3, N=4, and D=2.)

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \begin{bmatrix} 4 \\ 0 \\ 0 \\ 3 \end{bmatrix} = \begin{bmatrix} 1 \cdot 4 + 4 \cdot 3 \\ 5 \cdot 4 + 8 \cdot 3 \\ 9 \cdot 4 + 12 \cdot 3 \end{bmatrix} = \begin{bmatrix} 16 \\ 44 \\ 72 \end{bmatrix}$$

Notice how we can skip computations related to the entries of x that are equal to 0 since they cannot contribute to the result.

When working with sparsity, we use a *sparse encoding* to represent sparse data. This will allow us to compactly represent the non-zero parts of the vector without keeping storing all of the 0s, and it will allow us to build hardware that only performs arithmetic on the non-zero portions of the data.

To do so, we will use a simple format based on *Compressed Sparse Column* (CSC) encoding[1] to represent our sparse input vector x. In this encoding, only the non-zero entries of the vector are stored, but alongside of each value, we must store which *row* that it belongs to. For example, we would store the value of x in the example above as `val = [4, 3]` and `row = [0, 3]`. So, this tells us that the value 4 is in row 0, the value 3 is in row 3, and all other values are 0.

As another example, if N = 10, and x is represented by `val = [1, 2]` and `row = [5, 9]`, then this corresponds to a column vector with values `[0, 0, 0, 0, 0, 1, 0, 0, 0, 2]` (and D=2).

Compressing our sparse vector in this way obviously can make it smaller (if D is small), but it has another benefit: it allows hardware or software to perform computations while skipping the 0 entries. In pseudocode (where the matrix has M rows and N columns, and the vector, which has D non-zero entries, is encoded in the sparse formatted described above):

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

To make sure you understand this pseudocode, it is useful to work out the 3x4 example given above (where `val = [4, 3]` and `row = [0, 3]`).

Recall from above that a dense matrix-vector multiplication requires MN multiplications and $M(N-1)$ additions. Now, we can see that with a sparse vector with D non-zero entries, MSpVM requires only MD multiplications and $M(D-1)$ additions. If D is much smaller than N, this is a large reduction in the number of computations to perform. (E.g., if N = 1000 and D = 10, you have eliminated 99% of the computation.)

Your goal in this project is to build a hardware system that computes the product of a dense MxN matrix with a sparse vector. Your SystemVerilog code will be *parameterized* (using SystemVerilog

[1] When applied on matrices (instead of vectors), the CSC encoding is slightly more complex than this; in this project we simplify the representation because only our vector is sparse.

parameters) to allow the values of *M* and *N* to be easily changed in the code. The value of *D* will vary based on the vector you give your system as input. The following section introduces the specified structure of the design, its parameters, and the protocols it uses for input and output. Your system will take in a stream of values that represent a matrix and a sparse vector, compute the MSpVM, and output the result vector. Then your system will take in new inputs and repeat the process.

4. High-Level Project Overview

The goal of your project is to build a hardware system for MSpVM—that is, multiplications of dense matrices with sparse vectors. Figure 1 illustrates the top-level module and port specifications of the system. On the left are five signals whose names start with `INPUT_T`. These signals form an AXI-Stream interface that your system will use to receive input data. On the right there are three signals whose names start with `OUTPUT_T`. These from another AXI-Stream interface you system will use to transmit output data. A specification of the AXI Stream protocol for the inputs and outputs is provided in Section 5 below. There are also `clk` and `reset` signals. Assume `reset` is asserted high and synchronously applied.

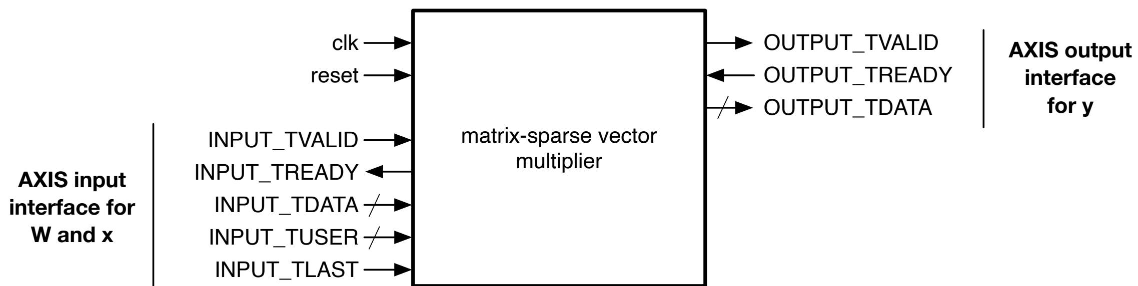


Figure 1. Top-level Design and Port Specifications.

Figure 2 illustrates a high-level block diagram of the system you will construct. Each of the components will be specified and described in more detail in the following sections.

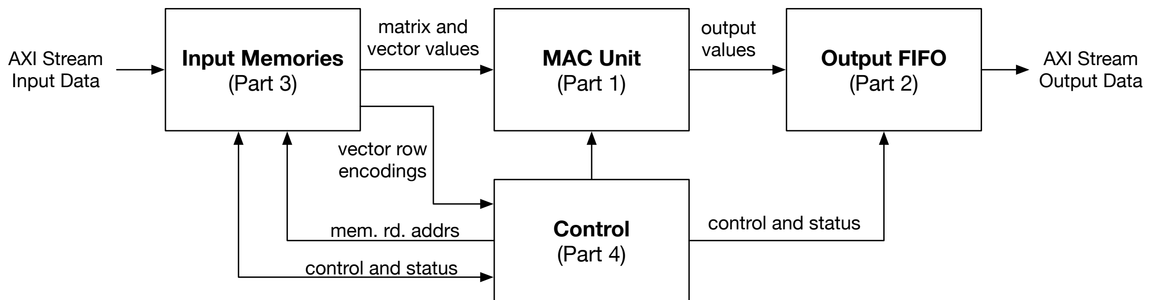


Figure 2. High-Level Block Diagram

- Your system will take as input a stream of data in AXI-Stream format that represents a matrix and a sparse input vector. (The sparse vector is encoded as described in Section 3 above.) Your system will perform a MSpVM of these and produce the result as output. The system's output values will be provided in AXI-Stream format. (The AXI-Stream protocol

and its use are described below in Section 5.) After completing a MSpVM, your system will accept a new set of inputs to compute. (In other words, your system will keep computing matrix-sparse vector products as long as new inputs are provided.)

- A multiply-accumulate (MAC) unit will be used to perform the individual multiplications and additions needed for the matrix-vector product. A MAC operation computes:

  ```
  f += a*b
  ```

  Take note of how this is the fundamental operation used in the matrix-vector produce pseudocode described above. The MAC unit is **Part 1** of the project, and it is described in the Project Part 1 document.

- As the MAC unit computes values of the output vector, it places them in the Output FIFO module, which is **Part 2** of the project. The Output FIFO module will buffer the values and output them from your system in AXI Stream format. This module is described in the Project Part 2 document.

- Your system will also require input memories to store the matrix and vector values while the system performs the computation. These are stored in the Input Memory module, which is **Part 3** of the project. This module will include a memory for the matrix, a memory for the vector, and necessary control logic. You can read more about this in the Project Part 3 document.

- The goal of **Part 4** of the project will be to integrate the three components from Parts 1–3 and design accompanying control logic that will allow the components to work together to perform the full matrix-sparse-vector product. The control logic will be responsible for coordinating the operation of the input memories, MAC unit, and output memories. You can read more about Part 4 in the Project Part 4 document.

- Lastly, the goal of **Part 5** of the project will be to optimize the speed of the system. Please see the Project Part 5 document.

Parameters

Rather than building hardware for a specific matrix size, you will design a *parameterized* system to allow flexibility in the matrix/vector dimensions and in the number of bits used for input and output values. This means that it will use the following SystemVerilog parameters:

- `M`: the number of rows of the matrix and rows of the output vector. Your system must support `M` ≥ 2.
- `N`: the number of columns of the matrix and rows of the input vector. Your system must support `N` ≥ 2.
- `INW`: the input bit width (the number of bits used per value in the input matrix and input vector). Your system must support 2 ≤ `INW` < 32 bits.
- `OUTW`: the output bit width (the number of bits used per value in the output vector). Your system must support 4 ≤ `OUTW` ≤ 64.
    - `OUTW` must also be large enough to prevent overflow. Please see explanation in the Project Part 1 document.

There is no defined upper bound on the limit of `M` and `N`, but as they get larger, the simulation and synthesis time will grow.

5. AXI-Stream Input/Output Protocol

Your system will use a slightly simplified version of ARM's AMBA AXI4-Stream protocol. (We will refer to our simplified version in this document as AXI-Stream or AXIS for short.)

AXI-Stream is shown in its simplest form in Figure 3. It is a synchronous protocol (meaning both sides share a common clock) that allows a transmitter[2] module to transfer data to a receiver when both sides "agree."

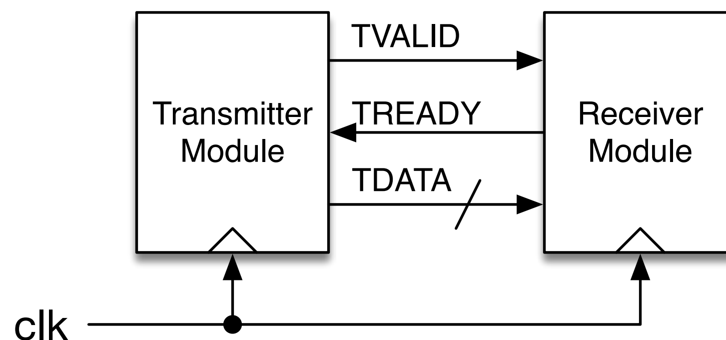


Figure 3. Simplified AXI-Stream protocol signals

The transmitter asserts the `TVALID` signal when it has placed valid data on the `TDATA` signal. The destination asserts the `TREADY` signal when it is capable of consuming that data. On any positive clock edge, data is transferred if (and only if) both the `TVALID` and the `TREADY` signals are asserted. (No data will ever be transferred unless *both* are asserted.) `TVALID` and `TREADY` are 1-bit signals, while TDATA is multiple bits.

---

[2] In earlier versions of the AXI standard, the transmitter was called a "master," and the receiver was called a "slave." This terminology was changed to "transmitter" and "receiver" in ARM's 2021 standard, although you will still see the older terms used in some places and CAD tools.

Note that both source and destinations modules must share a common clock. We will call this collection of signals (`TVALID`, `TREADY`, and `TDATA`) an "AXI-Stream interface." Figure 4 and Table 1 illustrate this functionality and timing. In this example `d[0]`, `d[1]`, etc., represent the data words transmitted.

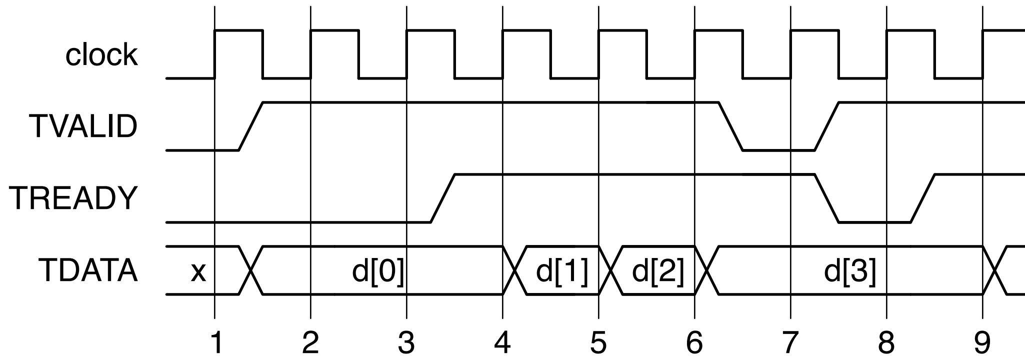


Figure 4. AXI-Stream data transfer timing example.

cycle #	TVALID	TREADY	Explanation
1	0	0	Neither valid nor ready; nothing is transferred
2	1	0	Transmitter puts data on `TDATA` signal and asserts `TVALID`. However, receiver module hasn't asserted `TREADY` so no data is transferred
3	1	0	Receiver module is still not ready (`TREADY==0`) so no data is transferred
4	1	1	Transmitter module is now ready (`TREADY==1`), so it receives data `d[0]`.
5	1	1	Since `d[0]` was transferred on the previous clock edge, the transmitter now changes the data to the next word. This word is transferred immediately. (Since `TVALID` and `TREADY` are still asserted.)
6	1	1	This has the same logic as cycle 5. Data word `d[2]` is transferred.
7	0	1	Now, the transmitter has de-asserted `TVALID`. The destination module does not read anything (regardless of what the source module has placed onto `TDATA`).
8	1	0	The transmitter has asserted `TVALID` but the receiver has de-asserted `TREADY`. Nothing is transferred here.
9	1	1	Both `TVALID` and `TREADY` are asserted, so the receiver reads `d[3]` from `TDATA`.

Table 1. AXI-Stream data transfer timing example.

The interaction of the TVALID and TREADY signals is called a *handshake*. Think of asserting TVALID as the transmitter holding out a hand; think of asserting TREADY as the receiver holding out a hand. If both sides hold out their hand, then they shake hands and agree that a data transfer is complete.

In our simplified AXI-Stream protocol, the transmitter is not permitted to wait until TREADY is asserted before asserting TVALID, and the receiver is not permitted to wait until TVALID is asserted before asserting TREADY.[3] In other words, both the transmitter and the receiver need to decide independently whether to assert their signal. (Then at the positive clock edge, they each *check* to see if the other side has asserted theirs.)

Additional AXI-Stream Signals

In addition to TDATA, TVALID, and TREADY, the AXI-Stream protocol includes several other control signals. In this project, you will use two of them: TUSER and TLAST, shown in Figure 5.

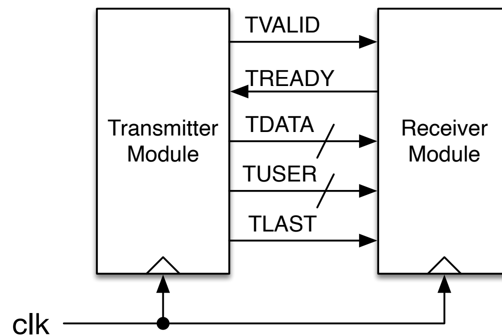


Figure 5. AXI-Stream signals including TUSER and TLAST.

- TUSER is a multi-bit signal that transmits "sideband data" from the transmitter to the receiver. Think of this as extra information that we transmit alongside of TDATA. This signal is controlled in exactly the same way as TDATA: Anytime TREADY and TVALID are 1 on a positive clock edge, then the information on TUSER is also transmitted.

- TLAST is a 1-bit signal that the transmitter can use to indicate that the currently transmitted data is the end of a transfer. Like TDATA and TUSER, this signal will be ignored except when TREADY and TVALID are asserted on a positive clock edge.

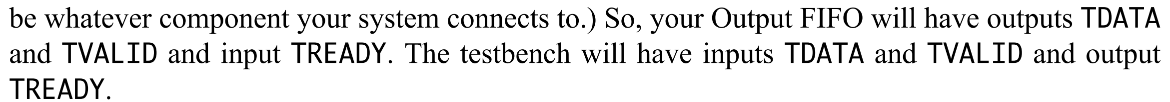
Output Stream

Your system will utilize TDATA, TREADY, and TVALID for its output. (For output data, you will not use TUSER or TLAST.) Your Output FIFO module (Part 2) will serve as the transmitter. When you simulate, the testbench will be the receiver for this output data. (In a real system, the receiver would

---

[3] One small difference between the full AXI-Stream protocol and our simplified version is that in the complete AXI-Stream Protocol, the receiver may choose to wait until TVALID is asserted before asserting TREADY, although we will not allow it in our project.

Another difference between our simplified AXI-Stream and the full protocol is that in the full protocol, once the transmitter asserts TVALID, it must keep it asserted until the handshake occurs; we will allow it to be de-asserted at any time.



be whatever component your system connects to.) So, your Output FIFO will have outputs `TDATA` and `TVALID` and input `TREADY`. The testbench will have inputs `TDATA` and `TVALID` and output `TREADY`.

Input Stream

Your system's input interface will use all five signals (including `TUSER` and `TLAST`). Your Input Memory module (Part 3) will serve as the receiver. When you simulate, the testbench will be the transmitter for this input data. For details about how the input data are provided in this stream, and how `TUSER` and `TLAST` are used to transmit sparse vectors, please see the Project Part 3 document.

6. Code and Report Submission

5 points will be awarded based on the quality of your code, comments, and report.

1. **Code**
   For your code and synthesis reports, you will turn in a single **.zip, .tar, or .tgz** file to Brightspace. ***Do not use a different archive format (e.g., .rar). Seriously, please do not use any archive format except .zip, .tar, or .tgz or you will lose points.***

   This compressed file should hold all of the code and synthesis reports from your project, organized into `part1/` through `part5/` directories. I will be testing your designs using my testbenches, so it is very important that your code sticks to the specification closely. I will test your designs using the ECE grad lab computers so make sure everything runs correctly ***there***.

   Do not turn in things like QuestaSim "work" directories or gate-level Verilog produced by synthesis. Please only submit your actual code.

2. **Synthesis Reports**
   Include the DesignCompiler synthesis report (in plaintext format) for each design you synthesized. These should be included in the .zip, .tar, or .tgz archive file mentioned above. Make sure these reports are clearly labeled. Please include them in the appropriate `part1/` `part2/` `part3/` `part4/` or `part5/` directory.

3. **Report**
   Please organize your report neatly. Use headings to separate it into Part 1, Part 2, Part 3, Part 4, and Part 5. Each part of the project will have a numbered list of questions you should answer. In your report, please use the same numbering to make it easier to find your answers. (In other words, number your answers to match the questions in this assignment.)

   In addition to the code submission, your report should be submitted **(as a PDF file only)** alongside of your.zip, .tar, or .tgz archive. (Please include the PDF report separately from the archive.) **If you worked with a partner, make sure you answered the questions in each Part where you are asked to explain each partner's contribution to the project.** (If you worked alone obviously you can skip this.)

4. **Electronic Hand-in Process**
   To hand in your code, go to Brightspace → Assignments → Project. There you can upload your `.zip`, `.tar`, or `.tgz` file and your PDF report. Only one partner should hand in for the group, but make sure both partners' names are clear in your code and report.

To create a .tgz file in Linux, first assemble a hand-in directory with copies of all of your code, etc. For this example, let's assume that directory is called `handin/`. Now, assuming you are one directory above `handin/`, type the following:

```
tar cvzf myhandin.tgz handin/
```

This will create a gzipped-tar file (`.tgz`) that contains the entire `handin/` directory (including all of its contents).

You can test that it worked properly by copying the `.tgz` file you created to another directory, and typing:

```
tar xvzf myhandin.tgz
```

This will extract the file into the directory you are currently in. If you have any problems with this or anything else, please post them on Piazza.

Please, only use .zip, .tar, or .tgz files for your archive, and use PDF for your report. If you use other formats, I will be unable to open your work on the lab computers, and you will lose points. Your code archive should only contain your code and your synthesis reports with clearly labeled names. Please do not submit the testbenches that were provided to you or other things like QuestaSim "work" directories or gate-level Verilog produced by synthesis.

Part 1: Multiply-Accumulate Computation Unit

Multiply and accumulate (MAC) is a basic operation used commonly in many different types of computations. MAC is defined as

$$f = f + a*b,$$

where *a* and *b* are inputs to the system and *f* is the output. Notice how the MAC operation is used in matrix-vector multiplication; as seen in the MSpVM sudocode (see also Section 3 Project Overview):

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

Your MAC module will have the following ports:

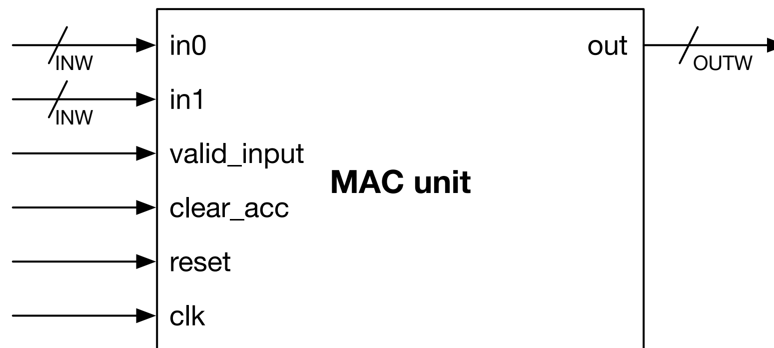


Figure 1.1. Top-level view of MAC unit's input and output ports

- Input signals `in0` and `in1` correspond to *a* and *b* in the equation above. Each will be signed values that are `INW` bits wide (where `INW` is a parameter in your SystemVerilog module).
- 1-bit input signal `valid_input` will be used to tell the module when a new valid set of `in0` and `in1` are provided on the input ports. The MAC unit will use the `valid_input` signal to determine when it should update the value of its internal accumulator register.
- Input signal `clear_acc` is used to clear the accumulator register that stores the value of the output. This is used to begin a new calculation.
- The output signal `out` corresponds to *f* in the equation above. This will be a signed value with `OUTW` bits (where `OUTW` is a parameter in your SystemVerilog module).

Obviously, a hardware system that performs this operation will require feedback because *f* depends on the previous value of *f*. (This is called "accumulation.") The *f* value will be stored on a register called the *accumulator*.

Your first task will be to implement and test a single-cycle unpipelined version of the MAC unit that looks like this:

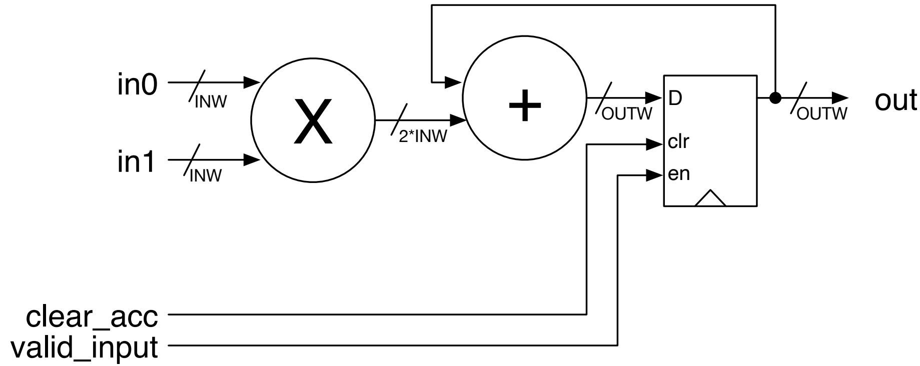


Figure 1.2. Unpipelined MAC unit.

Clock and reset signals are omitted in this figure but will be needed in the design. Assume that `in0` and `in1` are each `INW`-bit signed values. The output of the multiplier is a `2*INW` bit signed value, and the output signal and the adder output are signed `OUTW` bit signals. Assume the reset is positive-asserted (that is, when reset==1, the registers reset to 0). Assume the register resets synchronously—it resets when the reset signal is equal to 1 on a positive clock edge. (This will be true of all registers in your project.)

The accumulator register has two synchronous control inputs—`clr` and `en`. When `clr` is asserted on a positive clock edge, the register clears to 0. Otherwise, when `en` is asserted on a positive clock edge, the value on the register's input is stored. These two control inputs can be connected directly to the module's `clear_acc` and `valid_data` signals, respectively.

Overflow can occur when the result of an arithmetic operation cannot be represented using the allotted number of bits. Larger values of `OUTW` will allow the system to accumulate more values without overflowing. For example, you can tell that an adder has overflowed if:

- you add two positive values and get a negative answer, or
- you add two negative values and get a positive answer.

In this module, overflow can happen if the sum being calculated by the adder grows too large to fit in the `OUTW` bits available. In the case of overflow, your system should simply preserve the bottom `OUTW` bits of the value. (That is, if the adder overflows, you don't need to do anything special—just let it overflow.)

Store all of your files for part1 in a subdirectory called `part1/` and please make sure to use the file names and module names specified below.

Part 1.1: Your first task for Part 1 is to implement this system, simulate it, and synthesize it, finding the maximum clock frequency. Details about our provided testbench are below.

Implement this design in a file called `mac_unpipe.sv` and use the following top-level module name, port names, and port declarations:

```
module mac_unpipe #(
        parameter INW  = 16,
        parameter OUTW = 64
    )(
        input signed [INW-1:0]         in0, in1,
        output logic signed [OUTW-1:0] out,
        input clk, reset, clear_acc, valid_input
    );
```

It is very important that your module matches this input/output specification exactly, or it will fail the tests our testbench performs.

Simulate your design with a variety of parameter values using the testbench as described below. It should pass simulation for $2 \leq$ `INW` < 32 bits and $4 \leq$ `OUTW` ≤ 64. (Obviously, you don't want to exhaustively check all possible combinations of these, but you should try several combinations and check the extreme values.)

Part 1.2: Your second task for Part 1 is to pipeline this system by inserting a register between the multiplier's output and the adder's input, as shown in Figure 1.3. If pipelined correctly, you will be able to reach a higher clock frequency because the pipeline register splits up the critical path. See Topic 7 of our class for a more comprehensive discussion of pipelining. In your full matrix-vector multiplier (which you will create in Part 4), you will use this pipelined MAC unit (Figure 1.3).

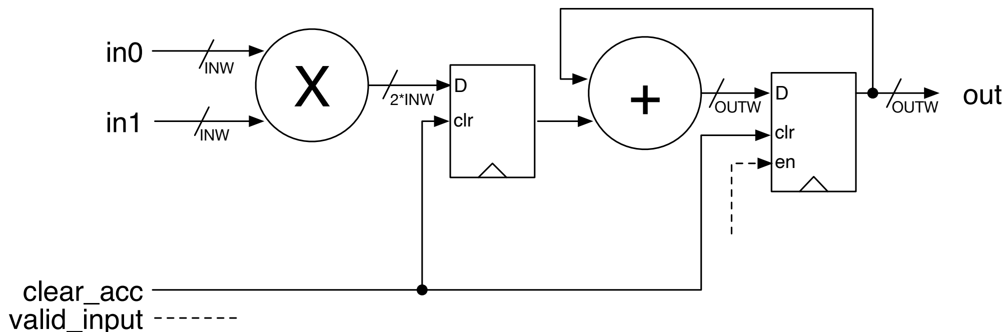
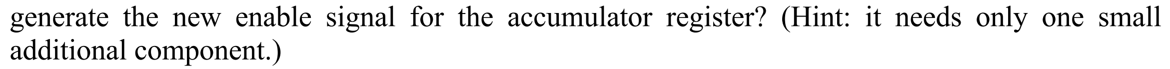


Figure 1.3. Pipelined MAC unit (but the accumulator's enable signal is not fully specified)

In your pipelined system, assume that `clear_acc` causes both the accumulator register and your new pipeline register to clear to 0.

Notice that this diagram doesn't show how the accumulator's enable signal is controlled. One small challenge here is that the enable becomes slightly more complex. You can no longer directly connect `valid_input` to the enable like in the unpipelined design, because it now takes longer for data to get from `in0` and `in1` to the register's input. Think carefully: how should your system



generate the new enable signal for the accumulator register? (Hint: it needs only one small additional component.)

Implement this design in a file called `mac_pipe.sv` and name the top-level module `mac_pipe`. Otherwise, use the same parameters and port declarations as in the unpipelined version.

Simulate your design with a variety of parameter values (like in Part 1.1) using the testbench as described below.

Testbench

You are provided with a testbench to test both `mac_pipe` and `mac_unpipe`. The testbench uses a parameter to choose whether you are simulating the unpipelined or the pipelined MAC. This testbench has a SystemVerilog module and an accompanying C program that the testbench runs via DPI.

You can find the testbench in the following two files:

```
/home/home4/pmilder/ese507/proj/part1/mac_tb.sv
/home/home4/pmilder/ese507/proj/part1/mac_tb.c
```

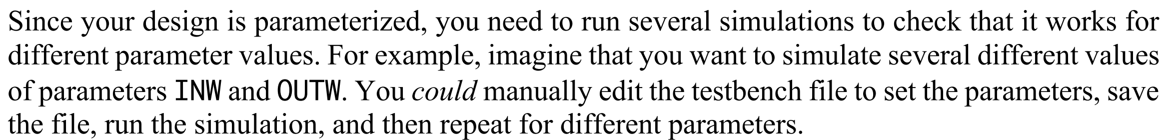
Copy these files into your `part1/` work directory where your Part 1 designs are.

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.

- I recommend beginning by reviewing slides 28–33 of the Topic 5 slides, which cover a testbench with a similar structure.
- The testbench has four parameters. Setting the `INW` and `OUTW` parameters in the testbench will set the corresponding parameters in the MAC module. These are used when the testbench instantiates your `mac_pipe` or `mac_unpipe` module.

  The `TESTS` parameter chooses the number of random tests to run.

  The `PIPELINED` parameter tells the testbench whether to use your unpipelined design (if `PIPELINED==0`) or your pipelined design (if `PIPELINED==1`). Make sure you set this parameter to 0 when simulating `mac_unpipe` and set it to 1 when simulating `mac_pipe`.
- The testbench uses a SystemVerilog class called `testdata` to hold one cycle of randomly generated test data. When a `testdata` object is randomized, it randomly chooses values of `in0`, `in1`, `valid_input`, and `clear_acc`.
- The C program contains two functions which simulate the expected behavior for both versions of the MAC. After generating a cycle of random test data, the testbench will call the appropriate C function to determine the expected result. Then, it checks that the simulated MAC output matches the expected output.
- Recall that it is possible for the MAC's accumulator to overflow if its value grows too large. The testbench accounts for this by simulating the overflow behavior in the expected results. In other words, if your inputs will make the MAC overflow, the testbench will *expect* it to overflow and check that it overflowed to the expected value.



Since your design is parameterized, you need to run several simulations to check that it works for different parameter values. For example, imagine that you want to simulate several different values of parameters `INW` and `OUTW`. You *could* manually edit the testbench file to set the parameters, save the file, run the simulation, and then repeat for different parameters.

However, there is one useful trick to make this easier—it is also possible to set those values at the command line when you run `vsim`. The following example will show how to do that.

- First compile the code. For the unpipelined design:
  ```
  vlog -64 +floatparameters +acc mac_tb.sv mac_tb.c mac_unpipe.sv
  ```

  or for the pipelined design:
  ```
  vlog -64 +floatparameters +acc mac_tb.sv mac_tb.c mac_pipe.sv
  ```

  (If your design uses any other .sv files, include them here also).

  Or if you just want to compile all the .sv files in your directory, you can run
  ```
  vlog -64 +floatparameters +acc mac_tb.c *.sv
  ```

  The `+floatparameters` option is necessary for QuestaSim to allow you to give parameter values at the command line when you run `vsim.`

- Then, you can run QuestaSim and set the parameters from the command like this:
  ```
  vsim -64 -c mac_tb -G INW=11 -G OUTW=44 -G PIPELINED=0
        -sv_seed random -do "run -all; quit"
  ```

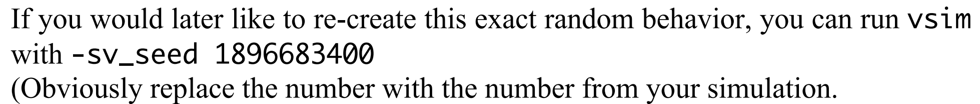
  - `-64` is necessary for the testbench to use DPI, which is required in this testbench. You will need to use `-64` each time you run `vlog` or `vsim` using this project's testbenches.

  - `-c` will run QuestaSim in command line mode. If you want to use the GUI to view waveforms, omit `-c`

  - The parts that start with `-G` like `-G INW=11` are used to set values of `mac_tb`'s parameters. The example above shows how to simulate with `INW=11`, `OUTW=44`, and `PIPELINED=0`. Change these values to match the simulation parameters you would like.

    - Alternatively, you could choose to edit the parameters in the module in mac_tb.sv and then skip all the `-G` parts of this command.

  - The `-sv_seed random` flag is used to initialize QuestaSim's random number generator, so you will get different random behavior each time you run it. When you simulate with this, the simulation will start by showing you the random seed it chose in a line that looks like the following (although obviously the number will be random)
    ```
    # Sv_Seed = 1896683400
    ```



If you would later like to re-create this exact random behavior, you can run `vsim` with `-sv_seed 1896683400`
(Obviously replace the number with the number from your simulation.

Report and Code Submission

After implementing and simulating both designs, complete the following tasks. In your report, provide the requested information and answer the following questions.

1. Use Synopsys DesignCompiler to synthesize your unpipelined design with `INW`=16 and `OUTW`=64 for a range of different clock frequencies from slow to fast. Adapt the scripts you used in HW2.

   Rather than copying them from your Homework 2 files, please re-copy the synthesis scripts from

   ```
   /home/home4/pmilder/ese507/synthesis/scripts/
   ```

   because I have updated them slightly since you completed homework 2. Don't forget to configure the script with your top-level module name, clock frequency, and so on. If your design is uses multiple `.sv` files, please see the instructions in the comments on lines 9 and 10 of `runsynth.tcl`. Do not include testbenches when synthesizing.

   Make sure you find the fastest possible frequency and save the synthesis report from that frequency in a plaintext file. Submit this clearly labeled synthesis report (as a plaintext file) with your code. I recommend naming it `mac_unpipe_synth.txt` and using a similar filename convention for future synthesis reports.

   For each frequency you try, record the area, power, the critical path location, and whether the timing constraint was met or violated. In your report, make a table that shows this data for each attempted frequency. Make sure you include units on all values you report (here and everywhere else in the report).

   Make graphs that show the relationships you found between clock frequency and both area and power. Explain the trends that you observed and explain why they occur. (Make two graphs. On both, show clock frequency on the x-axis; then show area as the y-axis on one graph and power as the y-axis on the other.) Make sure use graphs that plot both axes proportionally (like a scatter graph, not a line graph). Only include the design points where the timing constraint is MET.

   For each frequency, give a description of where the critical path is. Don't just copy/paste the endpoints from the synthesis report, but explain logically where the critical path lies in the module.

   It is very important that you correct any synthesis problems reported by DesignCompiler. If you have errors, the tool's output will not be correct. You also must be certain to fix any inferred latches from your design.

One common synthesis warning that you can safely ignore is:

```
Warning:   ./file.sv:182: unsigned to signed assignment occurs.
(VER-318)
```

If you have questions about other errors or warnings, first follow the instructions from the end of Topic 4 to use the "man" command in DesignCompiler to read the corresponding manual entries.

2. Now, repeat the tasks from question 1 for your pipelined design with the same values of `INW` and `OUTW`. Additionally, answer the following: Did pipelining help make this module faster? Explain why or why not and show how this is reflected in the synthesis data and critical path.

3. For the pipelined design with the maximum clock frequency you found, how much energy would your system consume if it were to process a sequence of 50 cycles of input values? Assume you have to wait until the final output comes out of the system.

   Remember: energy is measured in joules. Power = energy per time. 1 Watt = 1 Joule / 1 second. Use the power obtained from synthesis and your understanding of the time it would take for your system to fully compute 50 cycles of input values.

4. Would the energy you computed in question 3 change if you change the clock frequency? Justify your answer.

5. Make a table that compares the power, area, latency, and throughput of your pipelined and unpipelined MAC designs with `INW`=16 and `OUTW`=64. In your report show how you calculated the latency and throughput. Quantify latency in seconds (or ns), and quantify throughput in terms of MACs per second. (If needed, review these concepts in Topic 7.) Based on the trade-offs seen in your table, explain when it would make sense for a designer to choose the pipelined design and when it would make sense to use the unpipelined design.

6. Your design is pipelined as much as possible if you assume that you cannot pipeline the arithmetic units themselves. However, as we saw in Topic 10, we can also use DesignWare's pipelined arithmetic units, which can add pipeline stages inside of a multiplier. For example, you can replace the multiplier with one that is pipelined into 2, 3, 4, 5, or 6 stages. Based on your results to questions 1 and 2, would you expect that deeper pipelining in the multiplier may help? Justify why or why not. If you were to pipeline the multiplier deeper, what other changes would you have to make in your module? Would pipelining the adder be a good idea? Why or why not?

7. In questions 1 and 2, you always synthesized using the same values of `INW` and `OUTW`. Here, explore how changing those parameters affects the maximum possible clock frequency of your pipelined MAC module. Don't forget: to change these parameters for synthesis, you should edit the default parameter values in your source code (`mac_pipe.sv`).

   a. First, set `OUTW`=32 and synthesize four designs with `INW`=8, 12, 16, and 32. For each one, determine the maximum clock frequency. Make a graph that shows how the clock frequency changes with `INW`.

b. Then, set `INW`=16 and synthesize four designs with `OUTW`=16, 32, 48, and 64. Make a graph that shows how the maximum clock frequency changes with `OUTW`.

8. The MAC's accumulator holds `OUTW` bits. If the value stored in the accumulator grows large enough, this can overflow—`OUTW` bits may not be enough to store the resulting number.

   Given values for `INW` and `OUTW`, we would like to find the maximum value *G* such that we can guarantee, no matter what the input values are, there is no way the accumulator can overflow within *G* cycles of inputs to the MAC unit.

   First, calculate the maximum value of *G* for `INW`=4 and `OUTW`=10. (Hint: what is the largest magnitude number you could produce on the multiplier's output? Then, how many cycles would it take for that number to produce an overflow in the accumulator?) Don't forget that our values are all signed integers.

   Next, derive a generalized expression for *G* in terms of `INW` and `OUT`.

9. If you worked with a partner, please carefully describe each partner's contribution to this part of the project. (If you did not work with a partner, skip this.)

In your project submission, include a `part1/` subdirectory that includes:

- Your SystemVerilog implementations of both MAC units (`mac_pipe.sv` and `mac_unpipe.sv`) plus other SystemVerilog files your design uses (if any).

- Clearly labeled synthesis output files for the fastest clock frequency designs you found in questions 1, 2, and 7.

- Do not include the testbenches that were provided to you with the project, and do not include any unnecessary files like `work/` or `work_synth/` directories or other `gates.v` or other files created by our tools. Please only submit your code and the requested synthesis reports.

Part 2: Output FIFO

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).

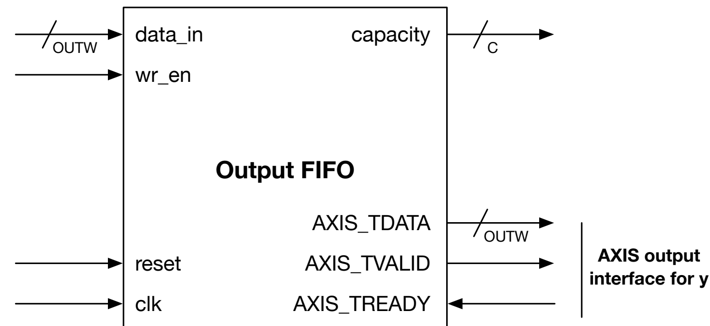


Figure 2.1. Top-level view of Output FIFO module.

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 ≤ `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 in class—see the Topic 8 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 class 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 class, 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 Topic 8, with synchronous reads and writes and two address ports. Use the following memory module. You can copy this module from:

/home/home4/pmilder/ese507/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.

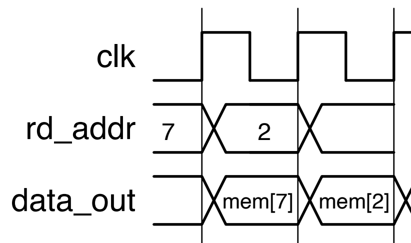


Figure 2.2. Timing of memory read.

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.

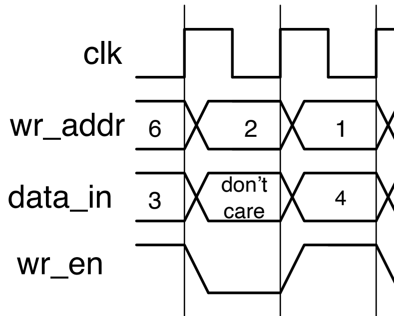


Figure 2.3. Timing of memory write.

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

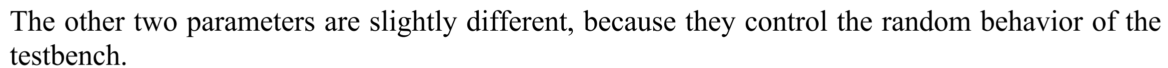
You are provided with a testbench to test your `output_fifo` module. This testbench will randomly write data into the FIFO and read data from it, checking the result. You can find the testbench at

`/home/home4/pmilder/ese507/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 class, 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 frequency. Save your synthesis report with a descriptive name and include it with your submission.

3. If you worked with a partner, please carefully describe each partner's contribution to this part of the project. (If you did not work with a partner, skip this.)

In your project submission, include a `part2/` subdirectory that includes your code and your synthesis report. Don't include any other files like synthesis work directories or other files created by the CAD tools.

Part 3: Input Memory Module

The goal of Part 3 is to construct and test the Input Memory module, which holds two memories, which will be used to buffer the input matrix and the sparse input vector. This module is parameterized by `M`, `N`, and `INW`. (See Project Overview Section 4 for the requirements on these parameters.) The following diagram shows the top-level block diagram of this module (Figure 3.1).

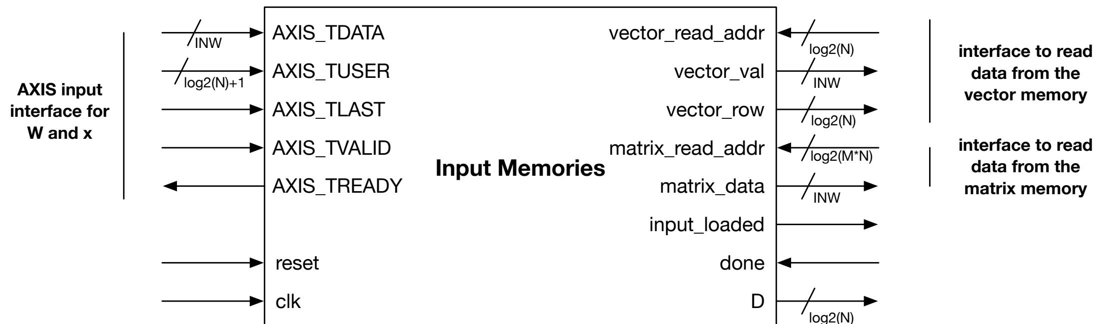


Figure 3.1. Input Memory module.

Internally, this module will contain control logic and two memories: one for holding the matrix *W* (called the *matrix memory*) and one for holding the sparse input vector *x* (called the *vector memory*) in the compressed sparse format described in Project Overview Section 3. On the left of Figure 3.1 you will see an AXI-Stream input interface, which will receive the input data (a matrix and sparse vector). On the right of Figure 3.1 you will see `vector_*` and `matrix_*` ports. These provide an interface that allows the rest of your MSpVM system to read the data stored in the input memories. Here is a specification of each port:

- `AXIS_TDATA`, `AXIS_TUSER`, `AXIS_TLAST`, `AXIS_TVALID`, and `AXIS_TREADY` collectively form an AXI-Stream interface used to load input data into the module. See Project Overview Section 5 for a description of AXI-Stream, and see the subsection labeled "Input Protocol" below for information on how this AXI-Stream interface will be used to load both input matrices and vectors. In your top-level MSpVM system, these signals will be directly connected to the top-level `INPUT_T*` signals (as shown in Figure 1 in Project Overview Section 4).

- The `input_loaded` signal is an output that your module will use to indicate when its internal memories hold a complete matrix and vector. We will refer to this as the "input_loaded" state. That is, when your module is in the input_loaded state, this signal will equal 1. (Later, your top-level MSpVM system will use this signal to determine when it is time to begin performing the computation.)

- `matrix_read_addr` and `matrix_data` are used to read data from the matrix memory. When your module is in the input_loaded state, then the `matrix_read_addr` signal will select an address in the matrix memory, and the matrix memory's output will be provided on `matrix_data`.

- `vector_read_addr`, `vector_val`, and `vector_row` are used to read data from the vector memory. When your module is in the input_loaded state, then the `vector_read_addr` signal will select an address in the vector memory, and the memory's output will be provided on `vector_val` and `vector_row` (indicating the values and rows for our sparse vector—see the explanation of CSC in Project Overview Section 3.3).

- `D` is an output that indicates the *D* value of the vector currently stored in the vector memory. (Recall from Project Overview Section 3.3 that *D* indicates the number of non-zero values in a sparse vector.) We require the `D` output to be correct and valid anytime `input_loaded` is 1.

- The `done` signal is an input that tells your module when it is done computing the MSpVM on the matrix and sparse vector values stored in memory. When `done` is asserted on a positive clock edge, your module should set `input_loaded` to 0 and go back to its initial state to begin taking in new input values.

Input Protocol

Matrix and sparse vector inputs will be provided to this module via the AXI-Stream interface signals shown on the left of Figure 3.1. Recall the discussion of AXI-Stream from Project Overview Section 5. Here there is some complexity, so we will break this description down into three parts: (a.) loading a matrix, (b.) loading a vector, and (c.) reusing an old matrix.

(a.) Loading a Matrix

When loading the matrix, your system will use the `AXIS_TVALID`, `AXIS_TREADY`, and `AXIS_TDATA` signals. If `AXIS_TVALID` and `AXIS_TREADY` are both equal to 1 on a positive clock edge, then your system has received data. Your system will control the value of `AXIS_TREADY`, so you must set it to 0 when the system is not "ready" for new inputs.

Matrix values will be provided in row-major order. This means that the first row will be provided, then the second row, etc. That is, the first valid input will be `W[0][0]`, then `W[0][1]`, …. Eventually after the end of the row (`W[0][N-1]`), then it will go to the next row `W[1][0]`, and so on. Your module must store the matrix values in the internal matrix memory in this order.

This process is illustrated in Figure 3.2 for a 3x3 matrix. Note that in this example, `AXIS_TVALID` and `AXIS_TREADY` are both 1, so the timing is simple. However, if either of these signals were to become 0, then nothing would be transmitted at that time, and the system must stall.

(Please also note in this figure we include a signal called `new_matrix`. This signal will be defined and explain in part (c.) below.)

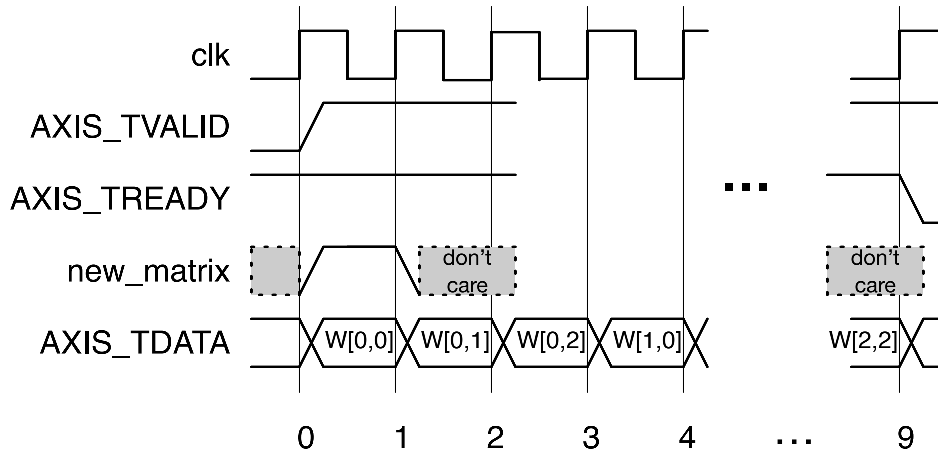


Figure 3.2. Example of loading the matrix.

(b.) Loading a Vector

After the matrix is transmitted, then the input stream will transfer the input vector. Loading the vector over the AXI-Stream interface is slightly more complex for two reasons:

1. We know that our vector will be stored in CSC format (see Project Overview Section 3.3.) This means that in addition to the `values` of the vector, we will also need the `row` indices. To transmit the `row` indices, we will use some of the bits from the `AXIS_TUSER` signal.

   We will define `AXIS_TUSER` as being `$clog2(N)+1` bits wide.
   - Its least significant bit, `AXIS_TUSER[0]` will be used for a different purpose. (We will discuss this in (c.) below.)
   - The remaining bits, `AXIS_TUSER[$clog2(N):1]` will be used to hold the `row` encoding of the element of the sparse vector. In the rest of this specification, we will refer to these bits as `row.`

   For simplicity, I suggest you make the following assignment in your SystemVerilog description of this module:

   ```
   logic [$clog2(N)-1:0] row;
   assign row = AXIS_TUSER[$clog2(N):1];
   ```

2. We also need to deal with the fact that the length of the compressed vector will be variable. That is, if *D* is 1, then only one input element (a pair of `val` and `row`) will stream into the system. However, if *D* is 100, then 100 input elements will stream in. To deal with the variable length problem, we will use the `AXIS_TLAST` signal. When loading the vector, if `AXIS_TLAST` is 1 on any positive clock edge (where `AXIS_TVALID` and `AXIS_TREADY` are both 1), then this indicates that this is the *last* value of the vector.

   For example, Figure 3.3 shows an example of first loading a 3x3 matrix, then a vector (where *D*=2). The hardware system only knows the vector is done when it sees the `AXIS_TLAST` signal.

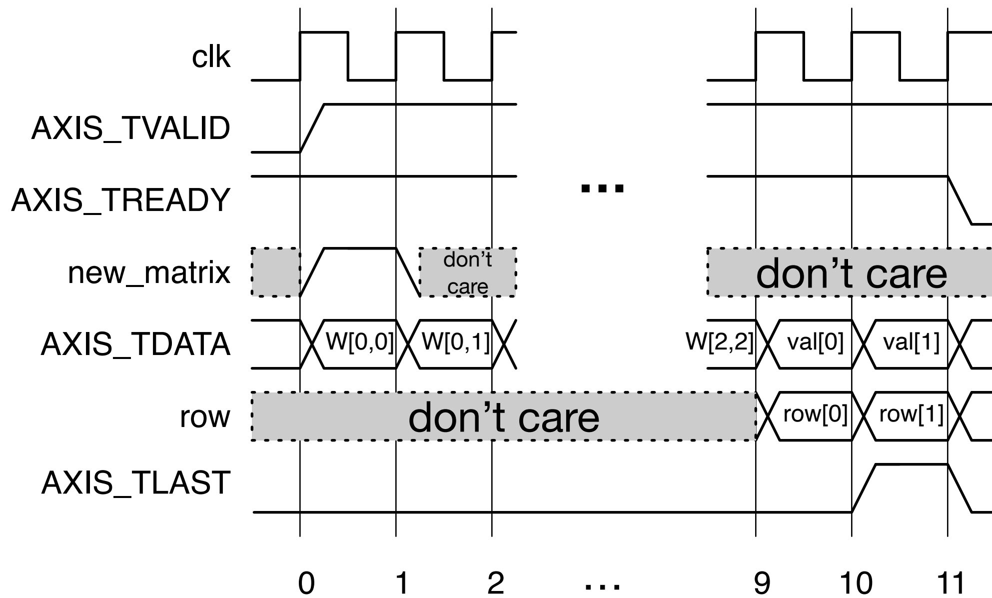


Figure 3.3. Example of loading a new 3x3 matrix, followed by a vector with *D*=2.

(c.) Reusing an Old Matrix

Lastly, your system has the capability of *remembering* the previous matrix, rather than loading a new matrix. This can allow operations where the system loads a matrix, and then multiplies that matrix with many different vectors without having to reload the matrix values as inputs. (Since the matrix can be large, this can save a lot of time.)

To make this possible, we need to define a control bit as part of the input stream to tell the system when it is loading a new matrix and when it should use the old one. For this we will use `AXIS_TUSER[0]`, the least significant bit of `AXIS_TUSER`. We will call this signal `new_matrix`.

For simplicity, you may want to make the following assignment in your SystemVerilog description of this module:

```
logic new_matrix;
assign new_matrix = AXIS_TUSER[0];
```

The `new_matrix` signal only matters during the first cycle of input data transfer. That is, as your system begins processing inputs for a new MSpVM, it will check `new_matrix` on the first valid input cycle. If `new_matrix==1` at that time, then the data represents the first value of the matrix. If `new_matrix==0` at that time, then the data represents the first value of the vector, and the system will continue to use the *old* matrix already stored in memory from the last operation.

The `new_matrix` signal is only meaningful during the first cycle of data transfer; at all other times, its value is ignored. Figure 3.3 above shows the process when `new_matrix==1`, so the input data represents a matrix and then eventually a vector. Then Figure 3.4 below shows an example where `new_matrix==0`, so the system skips reading the matrix and immediately reads a vector.

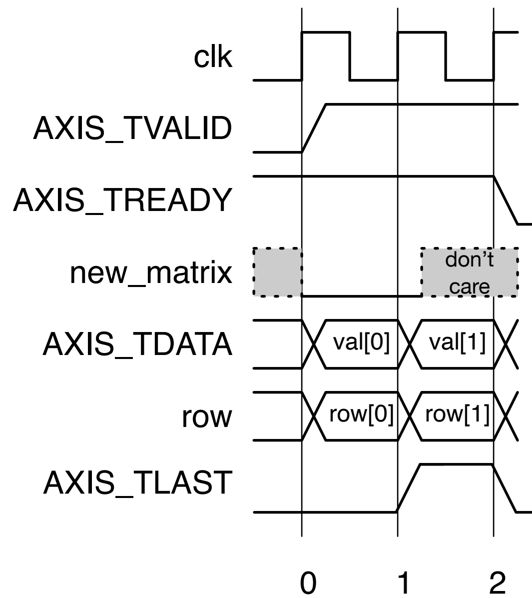


Figure 3.4. Example where `new_matrix==0` at the time of the first input, so a new matrix is not loaded, and the vector immediately begins.

Keep in mind that all transfers are done using the AXI-Stream protocol. This means that `AXIS_TDATA` and `AXIS_TUSER` are only transferred on positive clock edges when `AXIS_TVALID` and `AXIS_TREADY` are both equal to 1. Your system will set the value of `AXIS_TREADY`, so you must set it to 0 when the system is not "ready" for new inputs.

Module Operation

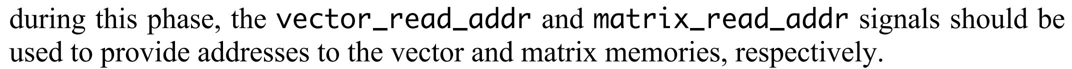
Here we will give a brief overview of this module's phases of operation and the sequence of steps it will take. You must construct control logic that will make the system undergo the following steps. Your control logic will likely need to include an FSM and counters. The following phases are not necessarily 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. *Input Matrix*: In this phase, the module will take in matrix data via its AXI-Stream input interface. This data will be stored in the matrix memory. This phase is complete after the entire matrix is loaded. This step will be skipped if `new_matrix` is 0. (See "Input Protocol" subsection above.)

2. *Input Vector*: In this phase, the module will take in sparse vector data via its AXI-Stream input interface. The data elements of the vector will be provided on `AXIS_TDATA` and the `AXIS_TUSER` signals, as described in the "Input Protocol" subsection above. Your system will monitor the `AXIS_TLAST` signal to know when to exit this phase.

3. *Input Loaded*: The input_loaded phase begins after the input vector is complete. Your system will output `input_loaded=1` during this phase. In this phase, your system should ensure that the `D` output signal holds the correct value of *D* (the number of non-zero elements in this vector).

   During this phase, your system will allow the vector and memory read interfaces (on the right side of Figure 3.1) to read data from the vector and matrix memories. This means that



during this phase, the `vector_read_addr` and `matrix_read_addr` signals should be used to provide addresses to the vector and matrix memories, respectively.

While in this phase, your system should monitor the `done` input signal. When `done==1` on a positive clock edge, your system should exit this phase, set `input_loaded` to 0, and go back to the beginning, waiting for new input data.

Memory

This module should contain two memory instances: one to hold the matrix and one to hold the encoded vector. Each memory must use synchronous reads and have a single address port. Use the following module for both memories. You can find this memory at:

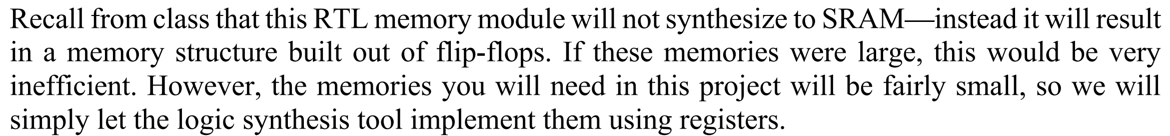
`/home/home4/pmilder/ese507/proj/part3/memory.sv`

```
module memory #(
        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]         addr,
        input                       clk, wr_en
    );

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

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

There are several important things to understand about this memory:

- The memory has one read port, one write port, and one address input used for both reads and writes. All reads and writes are synchronous (that is, they will occur on a positive clock edge).
- Unlike the memory used in Part 2, this memory module only contains a single address input, which will be used for both reading and writing. (So, you cannot read and write to different locations of this memory at the same time.)
- The memory's parameters are the same as in the dual port memory in Part 2.

The timing of reading and writing from this memory is the same as the dual port memory from Part 2 (Figures 2.2 and 2.3), except now there is a single address signal `addr` that is shared for both reads and writes.



Recall from class that this RTL memory module will not synthesize to SRAM—instead it will result in a memory structure built out 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.

- The matrix memory should have `WIDTH=INW` and `SIZE=M*N`. Your system should store the matrix in row-major order (see above).
    - The `data_in` port of this memory should connect directly to the `AXIS_TDATA` input of this module.
    - The `data_out` port of this memory should connect directly to the `matrix_data` output of this module.
    - The `addr` and `wr_en` ports of this memory should be driven by your module's internal logic.

- The vector memory should have `WIDTH=INW+$clog2(N)` and `SIZE=N`. In each memory entry, you should store the concatenation of the vector element's `val` (which is `INW` bits) and `row` (which is `$clog2(N)` bits). Each vector will need *D* entries, and *D* can be as large as *N*, so we must make this memory of size *N* to accommodate the largest possible vector.
    - The `data_in` port of this memory should be driven by a concatenation of the `AXIS_TDATA` input and the `row` signal.
    - The `data_out` port of this memory should connect directly to the `vector_data` and `vector_row` outputs of this module (with bits partitioned to match the way the value and row were concatenated on the memory's input port).
    - The `addr` and `wr_en` ports of this memory should be driven by your module's internal logic.

Reading from the memories

Once your module is in the input_loaded phase (and it is setting `input_loaded` to 1), your control logic should allow the memory read interfaces (on the right side of Figure 3.1) to read data from the matrix and vector memories. That is, when `input_loaded==1`, then the matrix memory's address input should follow `matrix_read_addr`, and the vector memory's address input should follow `vector_read_addr`. At all other times (when `input_loaded` is 0), your internal logic will determine the addresses on these memories.

Number of Non-Zero Vector Entries D

The number of non-zero values in the sparse vector is not known ahead of time—it is input-dependent. As described in the Input Protocol subsection above, your module will use the `AXIS_TLAST` signal to determine when you have reached the last of the input vector entries. Your module should contain a register to hold the value of *D*, and this register's output should go to the `D` output of this module. When your system determines that the entire input vector has been stored, it should set the correct value of *D* and hold it until new input data is loaded.

Code

Store all of your files for part3 in a subdirectory called `part3/` . Implement this design in a file called `input_mems.sv`. Your system should use the memory module given above. (You may either copy/paste that into your code or include the given `memory.sv` file alongside your file.) Use the following top-level module name, port names, and port declarations:

```
module input_mems #(
        parameter INW=16,
        parameter M=24,
        parameter N=32,
        localparam LOGN  = $clog2(N),
        localparam LOGMN = $clog2(M*N)
    )(
        input clk, reset,

        input [INW-1:0] AXIS_TDATA,
        input           AXIS_TVALID,
        input           AXIS_TLAST,
        input [LOGN:0]  AXIS_TUSER,
        output logic    AXIS_TREADY,

        output logic            input_loaded,
        input                   done,
        output logic [LOGN-1:0] D,

        input        [LOGN-1:0]  vector_read_addr,
        output logic [INW-1:0]   vector_val,
        output logic [LOGN-1:0]  vector_row,
        input        [LOGMN-1:0] matrix_read_addr,
        output logic [INW-1:0]   matrix_data
    );
```

Testbench

You are provided with a testbench to test your `input_mems` module. This testbench will generate random data (matrices and sparse vectors) and then provide the inputs to your module using the AXI-Stream interface (with random timing on the `AXIS_TVALID` signal). Then the testbench will wait until your module asserts the `input_loaded` signal, and then it will read the data back from your module's internal memories, checking that the data retrieved matches what was transmitted. You can find the testbench in the following two files.

```
/home/home4/pmilder/ese507/proj/part3/inputs_mems_tb.sv
/home/home4/pmilder/ese507/proj/part3/test_helper.c
```

Copy these files into your `part3/` work directory where your part 3 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.

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

- `INW`: the number of bits for each 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
- `TVALID_PROB`: a decimal value that represents the probability that the testbench will assert `AXIS_TVALID` on any given cycle. This should be between 0.001 and 1, inclusive. For

example, if you set TVALID_PROB to 0.3, then there will be a 30% chance that the testbench will assert AXIS_TVALID (and try to transmit input data) on each cycle. If you set this parameter to 0, the testbench will randomly pick a probability at the beginning of the simulation (and tell you what it chose).

The testbench uses a SystemVerilog class called testdata. This class is somewhat different than the one in Part 1's testbench. Here, the class holds the input data for an entire MSpVM operation. That is, it holds the values of the matrix, the vector (in sparse encoding), the value of *D*, and the value of new_matrix. The testbench uses SystemVerilog's internal randomization capabilities to randomize the values of the matrix, *D*, and new_matrix. Then, it uses a C function (called through DPI) to generate a sparse random vector.

For each test, the testbench will randomize the test data and feed the test data into the system via the AXI-Stream interface. The timing of this transaction will be randomized based on TVALID_PROB.

After feeding in a set of test data, the testbench will wait for the DUT to set input_loaded to 1. Then, the testbench will use the DUT's vector_read_addr and matrix_read_addr inputs to read the stored data back from the DUT's internal memories, checking the values.

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.)

Report and Code Submission

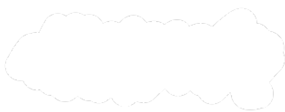
After implementing and simulating the designs with a variety of parameters, complete the following tasks. In your report, provide the requested information and answer the following questions.

1. This part of the project required you to design a significant amount of control logic that interacts with the AXI-Stream interface, the memories, the *D* register, and the input_loaded and done signals. Carefully and thoroughly document this module including your control logic. Your documentation should allow the reader to fully understand how your input_mems module works (and any submodules) without looking at the code.

2. The number of cycles required by this module is largely determined by the parameters (M, N), the input vector's D, and by how the testbench asserts AXIS_TVALID. However, there are places where you as the designer could make choices that affect the number of cycles required by your module. For example, if your system unnecessarily sets AXIS_TREADY to 0, or it adds extra cycles of delay between steps, the system will be less efficient.

   One way to quantify this is to measure how long your system takes to complete a task. Run a simulation where you set INW=16, M=24, N=32, and TVALID_PROB=1. When new_matrix==1, the testbench will begin by feeding in a 24*32=768 matrix values and *D* vector elements (where $1 \leq D \leq 32$). In other tests where new_matrix==0, the system will simply feed in *D* vector elements.

   Simulate this design in QuestaSim's waveform view and count the number of cycles between when your design sets AXIS_TREADY to 1, and when it sets input_loaded to 1.



Do this for a set of inputs where `new_matrix==1` and a set of inputs where `new_matrix==0`.)

Hint: you can view the amount of simulated time that passes in the waveform and then divide it by 10ns to get the number of simulated clock cycles. In your report, give this cycle count, and the value of D for the vector (for both `new_matrix` of 0 and 1).

In the report, quantify how efficient your system is with respect to the number of clock cycles by computing the following ratios:

- When `new_matrix==1`, use efficiency = cycles/(M*N+D)
- When `new_matrix==0`, use efficiency = cycles/D

In these metrics, 1.0 is perfectly efficient—a good implementation will be close to this. Report both metrics and the data you collected.

If your efficiency number is not close to 1, where could your logic be improved?

3. In the previous question, you measured the cycle count. Now, use your understanding of your system's behavior to write equations for the cycle count with respect to `M`, `N`, and `D`. You should have one equation for `new_matrix==1`, and one equation for `new_matrix==0`.

4. Describe how your system's hardware changes when you change the parameters `INW`, `M` and `N`. Be specific about how the hardware components in your design will change as you change these parameters. Which of these changes do you expect to be important to the area and power of the system? Explain your answers.

5. Synthesize the design using DesignCompiler twice, with parameters:
   - `INW`=16, `M`=4, `N`=5
   - `INW`=16, `M`=24, `N`=32

   Correct any synthesis problems you find. For each set of parameters, find the maximum possible clock frequency, area, power, and critical path location. (Here you only need to report statistics for the highest clock period for each design.) Carefully explain each design's critical path. Don't just list its start and end points; explain what the path means and why it makes sense. Does the critical path change between these two designs? Explain why or why not.

   Save both of your synthesis reports with descriptive names and include them with your submission.

6. If you worked with a partner, please carefully describe each partner's contribution to this part of the project. (If you did not work with a partner, skip this.)

In your project submission, include a `part3/` subdirectory that includes your code and your synthesis reports. Don't include any other files like synthesis work directories or other files created by the CAD tools.

Part 4: Matrix-Sparse Vector Multiplier (MSpVM)

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.

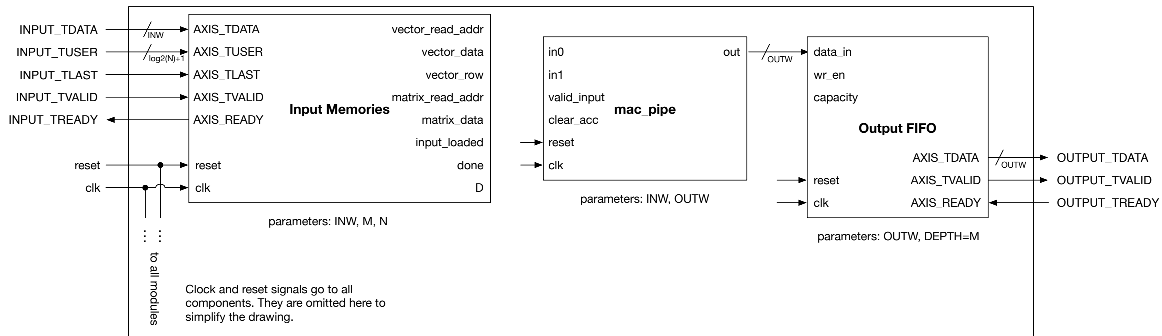
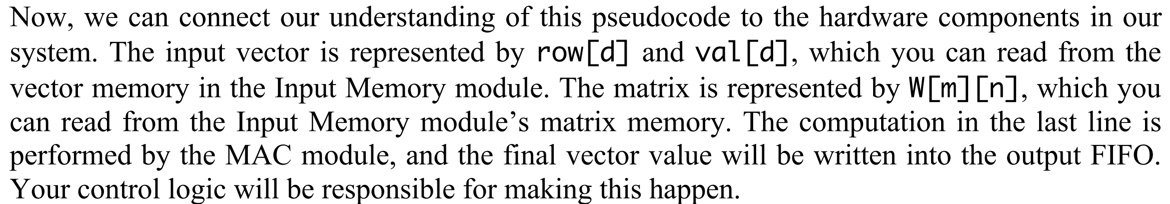


Figure 4.1. This shows a partial view of how your top-level `MSpVM` module should be constructed. You will need to design logic to interact with the signals that are unconnected in this diagram. (You may need to zoom in on the PDF to read this diagram clearly.)

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]`

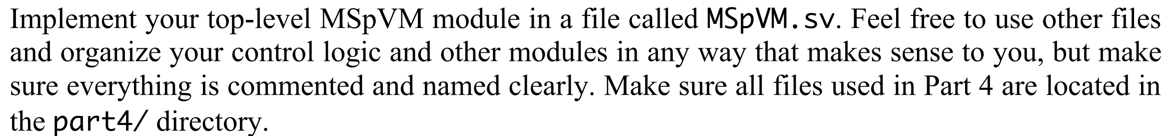
…
- Cycle `D-1`: read `W[m][row[D-2]]` while concurrently reading `row[D-1]` and `val[D-1]`
- Cycle `D`: read `W[m][row[D-1]]` from matrix memory

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
    );
```

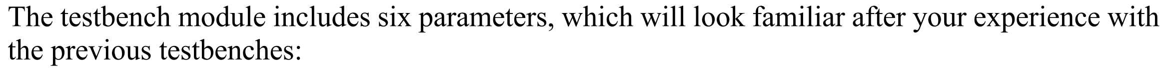
Testbench

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/pmilder/ese507/proj/part4/MSpVM_tb.sv
/home/home4/pmilder/ese507/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.

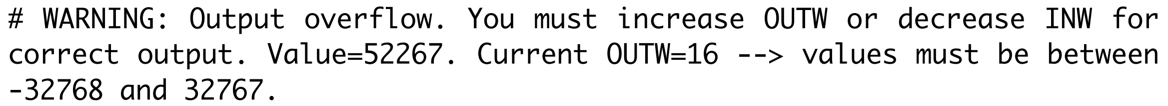
The testbench will check each output value and report any errors to the screen. The testbench will also report your system's simulated throughput in terms of MSpVMs per cycle. We will discuss this more in the "Throughput" subsection below.

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`)

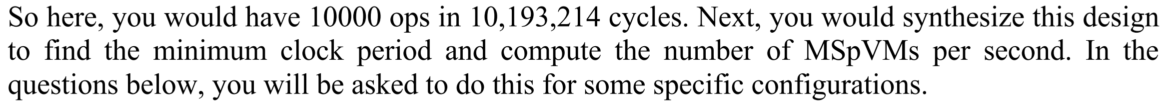
When we measure throughput, we will choose values for hardware parameters `M`, `N`, `INW`, and `OUTW`. Then we will choose values for testbench parameters `INPUT_TVALID_PROB` and `OUTPUT_TREADY_PROB`. We will measure the cycle count over many operations. `TESTS = 10000` is a good number. Although `D` 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).

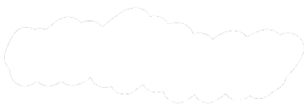
   Based on these equations, is your system's performance limited by any one phase of execution? (For example, if your input loading time is much higher than the compute time, this shows that the input loading time limits your speed much more than computation. On the other hand, if the system spends most 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.

8. If you worked with a partner, please carefully describe each partner's contribution to this part of the project. (If you did not work with a partner, skip this.)

In your project submission, include a `part4/` subdirectory that includes your code and your synthesis reports. Don't include any other files like synthesis work directories or other files created by the CAD tools.

Project Part 5: Performance Optimization

Part 5: Performance-Optimized MSpVM

In Part 5, you will optimize the throughput and delay of your full MSpVM system (from Part 4). You will modify your system to be as fast as possible, while keeping the input/output ports and protocols as specified in the prior parts of the project. Your goal is to maximize the throughput and minimize the delay of the system. We will quantify your optimizations based on the same metrics you used in Part 4 (throughput, delay, area, power, energy, area-delay product). Here you should aim to increase throughput and minimize delay, even at the expense of area, power, and energy.

Create a new directory `part5/` for this part. Be very careful that you keep your original Parts 1–4 files in their original directories. Don't accidentally lose your prior work here! I suggest that you start by copying all of your Part 4 design files into your Part 5 directory and modifying it from there.

You may change the internals of your system in any way possible, but the input/output ports and protocol must not deviate from the original specification. In other words, Part 5 your optimized system must simulate correctly with the exact same testbench as the Part 4 design.

Some ideas you may want to pursue:

- pipelining deeper, potentially including pipelined DesignWare components. An example of using DesignWare pipelined mulitpliers is given below.

- increasing parallelism. That is, building more adders, multipliers, and memories so that you can perform more operations concurrently.

- modifying your control logic so that the system works more efficiently. One potential improvement in this direction is overlapping the loading of input data with computing. For example, you could *double buffer* your input memories so that while your system is computing an MSpVM, your `input_mems` module is concurrently pre-loading the next set of inputs.

You may not modify the internals of the memory modules used (except of course adjusting their parameters when you instantiate them). However, you may choose to use as many memory module instances as you like.

There are obviously many techniques you can consider using here. Use your understanding of your system and the data collected in your evaluations from prior parts of the project to determine what techniques will give you the most improvement here. In Part 5 you will be evaluated based on the correctness of your optimized system and the extent to which you improved the performance. A portion of the points here will be directly calculated based on the speed of your system.

DesignWare Library Components

This subsection describes how to use DesignWare library components, especially pipelined multipliers. Do not interpret the inclusion of this text to mean that this is necessarily the most important optimization to make. The details are given here because they are needed for you to apply

this technique (while the other techniques do not require any specialized knowledge beyond what you already have).

You may use DesignWare components for adders and multipliers, but you may not use the DesignWare multiply-and-accumulate unit.

One useful component is the DesignWare pipelined multiplier. Recall, in Part 1, you pipelined your MAC unit by placing a register between the multiplier and adder. With that style of design, where you are using the * operator for multiplication, it is not possible to pipeline the multiplier itself into multiple stages. The DesignWare pipelined multiplier designs are internally-pipelined multipliers, which may be able to improve your clock frequency. We can *instantiate* a pipelined multiplier from their library.

You can find the DesignWare datasheet on Brightspace (under Course Documents → DesignWare Documentation), but here is a quick guide on how to instantiate the pipelined multiplier:

```
DW02_mult_S_stage #(INW, INW) multinstance(input1, input2, 1'b1,
      clk, output);
```

In this code, replace the `S` in `mult_S_stage` with the number of pipeline stages you want inside of the multiplier (from 2 to 6). Replace `input1` and `input2` with the two `INW`-bit signals you want to multiply. Replace `output` with the `2*INW`-bit signal where you want to store the multiplier's output.

For pipelined systems like these to work correctly, you will need to adjust your control logic. Think carefully about how your control logic must change for different choices of `S`. Make sure that your design continues to simulate correctly.

Note that once you use one of the DesignWare instantiations in your design, you need to take an extra step when simulating. When you compile your code for QuestaSim, you need to also compile the *simulation module* for the DesignWare designs. To do, simply include the multiplier's simulation model when you run `vlog`. These simulation models are:

```
/home/home4/pmilder/ese507/synthesis/sim_ver/DW02_mult*.v
```

Or, if you use other DesignWare components, replace the end of that line with the appropriate one. (You will not need to do anything special when trying to synthesize these modules because DesignWare components are already accessible to DesignCompiler.)

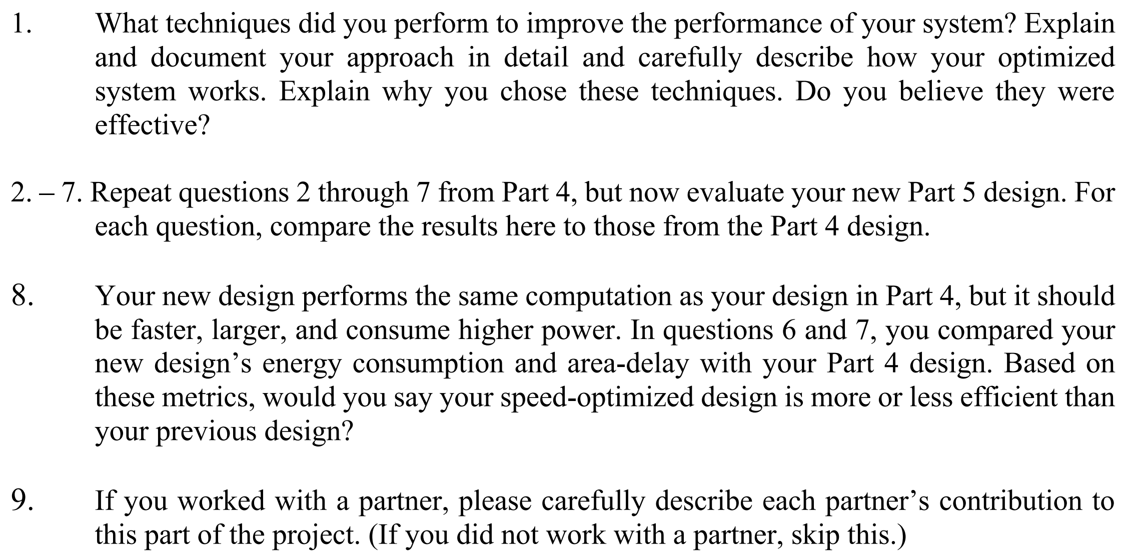
Code and Testbench

Store all of your files for Part 5 in a subdirectory called `part5/` . Use the same top-level module name and port specification as Part 4. Since your optimizations will not change the external input/output behavior, you will use the same testbench as Part 4.

As you improve your design, evaluate it based on the metrics described below.

Report and Code Submission

After implementing and simulating your Part 5 design with a variety of parameters, complete the following tasks. In your report, provide the requested information and answer the following questions.



1. What techniques did you perform to improve the performance of your system? Explain and document your approach in detail and carefully describe how your optimized system works. Explain why you chose these techniques. Do you believe they were effective?

2. – 7. Repeat questions 2 through 7 from Part 4, but now evaluate your new Part 5 design. For each question, compare the results here to those from the Part 4 design.

8. Your new design performs the same computation as your design in Part 4, but it should be faster, larger, and consume higher power. In questions 6 and 7, you compared your new design's energy consumption and area-delay with your Part 4 design. Based on these metrics, would you say your speed-optimized design is more or less efficient than your previous design?

9. If you worked with a partner, please carefully describe each partner's contribution to this part of the project. (If you did not work with a partner, skip this.)

In your project submission, include a `part5/` subdirectory that includes your code and your synthesis reports. Don't include any other files like synthesis work directories or other files created by the CAD tools.