Stony Brook University

Project 2: Matrix-Vector Multiplication

Issued: 10/4/17; ~~Due: 11/1/17~~ Extended to 11/6/17, 4:00PM

Introduction

In this project, you will utilize memories and the multiply-and-accumulate (MAC) units from Project 1 to build a system that performs matrix-vector multiplication. Later, Project 3 will build on this system to construct hardware for evaluating neural networks.

In part 1, you will construct a system that takes as input a matrix and a vector, and outputs their product. In part 2, you will extend it to additionally add a vector to the result. In part 3, you will expand this system to work with a larger input size. Then, in part 4, you will modify your system to make it faster.

The objective of this project is to give you more experience creating designs in SystemVerilog, testing them, and evaluating them through synthesis. Further, you will gain insight into how changes you make to a system will affect its costs and performance. You will turn in:

- your documented code including all testbenches for all four parts
- clearly labeled synthesis reports for each time you are asked to synthesize a design
- a report that answers specific questions asked throughout this assignment

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

1. Make sure the timing of all signals in your designs matches the specifications in this handout
2. Carefully label and document your code
3. Create separate subdirectories for each part of the project (part1, part2, part3, part4)
4. Include a README file in each directory giving a description of each file, and the exact commands you are using to run simulation and (where appropriate) synthesis.

Your project will be evaluated on the correctness and efficiency of your designs, the thoroughness of your testbenches, and your answers to the 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). All code will be run through an automatic code comparison tool.**

If you have general questions about the project, you may email me, or post them to Piazza.

If you would like a private Git repository for you and your partner to use for sharing code, please contact me and I will make an account for you on my server.

If you have general questions about the project please post them Piazza.

Deadline: This project was originally due Wednesday 11/1. We have extended the due date until Monday Nov. 6th at 4pm. No further extension will be given. This project is long and is significantly more complicated than Project 1. Please start early.

Partner

If you are choosing to work with a partner, by Monday 10/9 at 11:59pm you must:

- Send an email to peter.milder@stonybrook.edu with the subject "ESE 507 Project 2 Partner Signup"
- Send the email from your @stonybrook.edu email address
- In the body of the email, write both your name and your partner's name
- CC your partner on the email (using your partner's @stonybrook.edu email address)

After this, you are committed with working with this partner for project 2. (If you want to change for later projects, you may.)

Background

Matrix-Vector Multiplication (Part 1)

We first begin by reviewing matrix-vector multiplication. As an example, let W_3 represent a square 3×3 matrix, and let x represent a (column) vector of length 3. The product $y = W_3 x$ is defined as:

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

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).

More generally, if we have a system that takes in a $k \times k$ input matrix W_k and a length k input vector x, the system would thus take in k^2+k inputs and would produce k outputs.

Matrix-Vector Multiplication plus Vector Addition (Parts 2–4)

Next, we can add a vector addition onto the MVM. This is defined as:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} w_{0,0} & w_{0,1} & w_{0,2} \\ w_{1,0} & w_{1,1} & w_{1,2} \\ w_{2,0} & w_{2,1} & w_{2,2} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} \quad (2)$$

So, this system takes in 15 values (the 3×3 matrix W_3 and the 3×1 column vector x, and the 3x1 column vector b) and produces 3 values (3×1 column vector y).

More generally, if we have a system that takes in a $k \times k$ input matrix W_k, a length k input vector x, and a length k vector b, the system would thus take in k^2+k+k inputs and would produce k outputs.

Memory

This project will require the use of memories. The following is the SystemVerilog description of the memory you will use. (You can also find the code at `/home/home4/pmilder/ese507/proj2/memory.sv`)

```
module memory(clk, data_in, data_out, addr, wr_en);

    parameter WIDTH=16, SIZE=64, LOGSIZE=6;
    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. All reads and writes are synchronous (that is, they will occur on a positive clock edge).

- Unlike the examples we looked at in class, this 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 is parameterized by three parameters:
    a. `WIDTH`, the number of bits of each word
    b. `SIZE`, the number of words stored in memory
    c. `LOGSIZE`, the number of address bits needed to address `SIZE` entries (this is the log base two of `SIZE`, rounded up)

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

    ```
    memory #(12, 256, 8) myMemInst(clk, din, dout, addr, wren)
    ```

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

Figure 1 demonstrates the timing of reading from the memory. On each positive clock edge, the system samples the value on `addr`. A short time after the edge, it will output the value in memory at location `addr`. In this diagram, `mem[7]` just represents the value stored in address 7 of the memory.

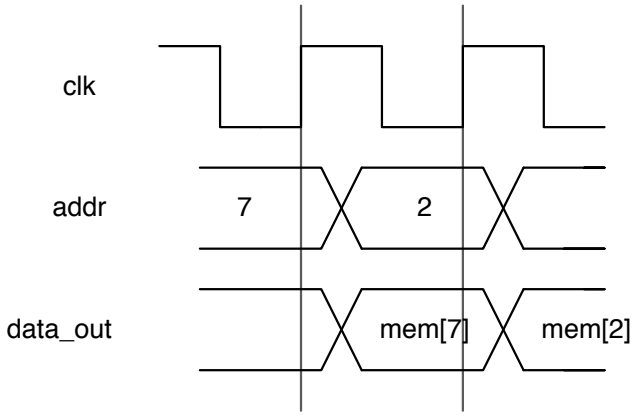

Figure 1. Timing of memory read.

Figure 2 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 in the third cycle.

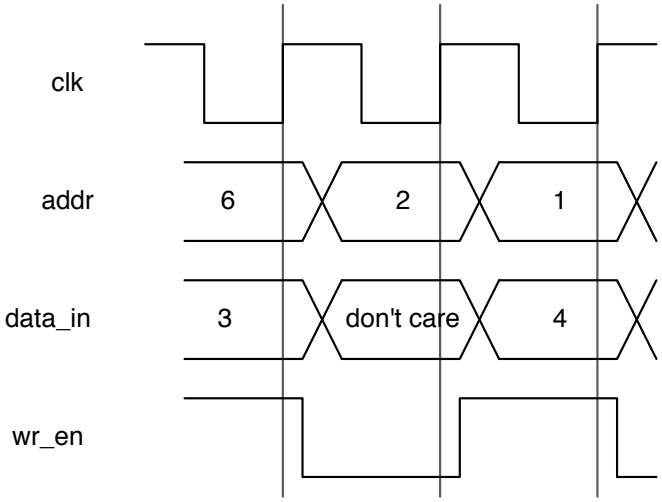

Figure 2. Timing of memory write.

Recall from class that this RTL memory module will not synthesize to SRAM—instead it will produce a memory structure 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 very small, so we will simply let the logic synthesis tool implement them using registers.

Input/Output Protocol

For this and future projects, we will use a simplified variation on ARM's AXI4-Stream Protocol. Shown in Figure 3, this is a simple synchronous protocol that allows a sender (called a "master") to transfer data to a receiver (called a "slave") when both sides agree.

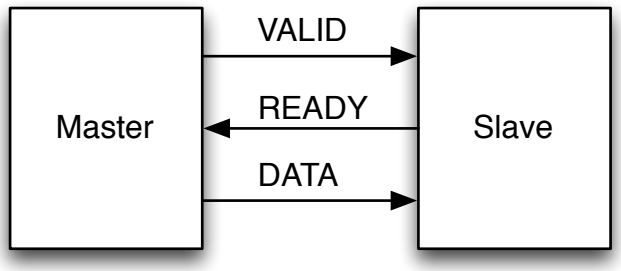

Figure 3. Handshaking signals

The master asserts the VALID signal when it has placed valid data on the DATA signal. The slave asserts the READY signal when it is capable of consuming that data. On any positive clock edge, data is transferred if (and only if) both the VALID and the READY signals are asserted. (Unless both are asserted, no data is transferred.)

The diagram (Figure 4) and accompanying table (Table 1) on the next page illustrate this process.

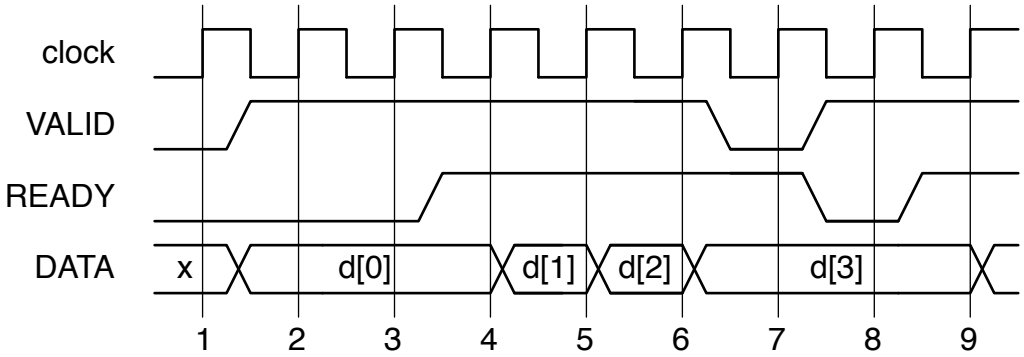

Figure 4. Handshake Timing Example

cycle #	VALID	READY	Explanation
1	0	0	Neither valid nor ready; nothing is transferred
2	1	0	Master puts data on DATA signal and asserts VALID. However, slave is not READY so no data is transferred
3	1	0	Slave is still not READY so no data is transferred
4	1	1	Slave is now READY, so it receives the data word d[0].
5	1	1	Since d[0] was transferred on the previous clock edge, the master now changes the data to the next word. This word is transferred immediately.
6	1	1	Same logic as cycle 5. Data word d[2] is transferred
7	0	1	Now, the master has de-asserted VALID. The slave does not read anything (regardless of what the master has placed onto DATA).
8	1	0	The master has asserted VALID but the slave has de-asserted READY. Nothing is transferred here.
9	1	1	Both VALID and READY are asserted, so the slave reads d[3] from DATA.

Table 1. Handshake Timing Example

Your system will use this "handshake" for its input and output signals. This means your system will receive inputs on a slave port and produce outputs on a master port. Its slave port must take as input a "valid" signal and ignore any invalid data, and it must output a "ready" signal that indicates when your system is ready for new inputs. This signal should only be 1 when your system is capable of reading in new data.

Similarly, your system will use a master port for outputs; your system will produce a "valid" that indicates when the system is producing valid output data, and it will take as input a "ready" signal. If this "ready" is not asserted, your system cannot transfer outputs and must wait.

Basic System Architecture

As you can see from expression (1) above, each row of the output vector y can be computed using a multiply-and-accumulate (MAC) unit, as you built in Project 1. For example, you would calculate y_0 by first clearing the accumulation memory, and then feeding the unit inputs $m_{0,0}$ and x_0 on the first cycle, $m_{0,1}$ and x_1 on the second cycle, and $m_{0,2}$ and x_2 on the third cycle.

In order to hold the input data (matrix M and vector x), we will be using memories, and we will use a control module to control the entire system.

Figure 5 shows the top-level view of your design. In addition to inputs `clk` and `reset`, the system has "valid" and "ready" signals used to control the input and output timing (see "Input/Output Protocol" above). Additionally, the system has an 8-bit data input port and a 16-bit data output port.

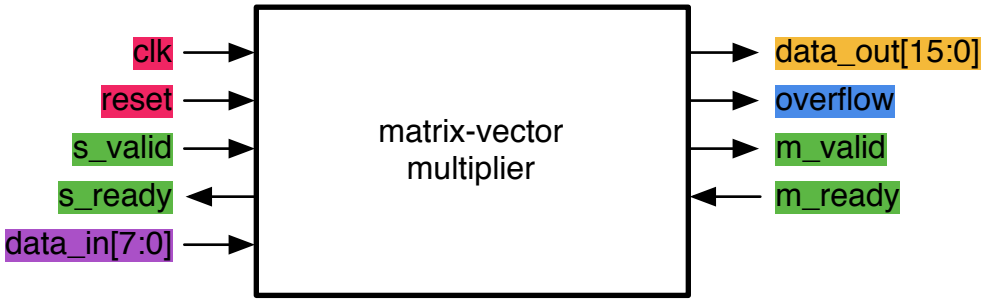

Figure 5. Top-level design. See below for information about valid/ready signals.

Your system will receive its input on a "slave" port, and it will produce its output on a "master" port.

After your system finishes a computation, it should be able to immediately begin taking a new set of inputs for the next computation. Remember, your system should only assert s_ready when it is capable of taking new input in. Similarly, your system should not be providing new output when m_ready is not asserted. If your system is ready to provide output but m_ready is not asserted, your system will need to wait.

Part 1: 3×3 Matrix-Vector Multiplier

In Part 1, you will get started by building a relatively simple system for k=3, that is, your system will multiply a 3×3 matrix with a vector of length 3.

Your system will use the Input/Output protocol described above. The inputs will be given in the following order: matrix M in row order, followed by vector x. That is, the first input provided will be $m_{0,0}$, the next will be $m_{0,1}$, and so on. After all nine matrix inputs are given, the next three inputs will be x_0, x_1, and x_2. Your system should output three output values y_0, y_1, and y_2.

If your system detects an overflow on any of the three output values, it should assert the "overflow" output during the time it outputs ***that value***. Note that this is ***different*** than the way overflow worked in Project 1. For example, if you are using your system to compute:

$$\begin{bmatrix} 10 & 11 & 12 \\ 127 & 127 & 127 \\ 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} 127 \\ 127 \\ 127 \end{bmatrix}$$

you would find that y1 should equal 48387, but this overflows, so your system will output the overflowed value and set overflow = 1 at the same time. However, y0 and y2 would both be computed correctly, meaning overflow should equal 0 while they are being outputted. (Hint: clear your "overflow" register each time you clear your accumulator.)

Here you will build a datapath based on the unpipelined MAC unit from Project 1, with some added memories. Figure 6 shows a rough idea of what this system should look like. Each block labeled `mem` is an instantiation of the module named `memory` from above. (Note that you will need to determine the correct parameters for these modules, as well as the bits needed for the address lines.)

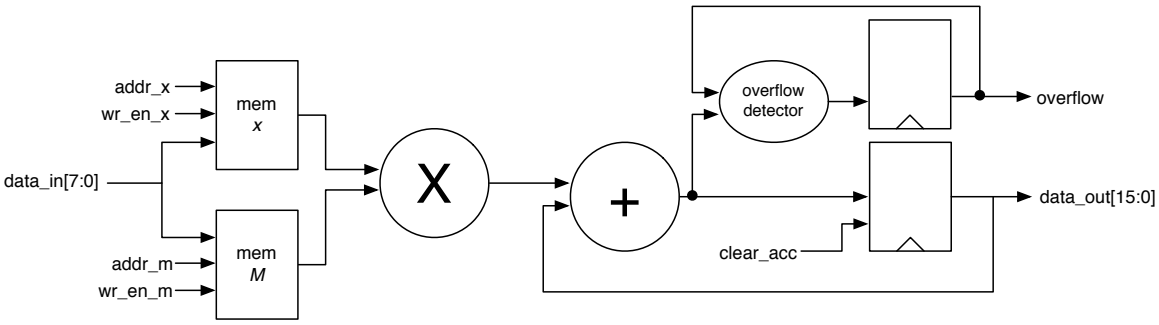

Figure 6. Datapath structure.

Note that many of the datapath's inputs are control signals. These signals (`addr_m, wr_en_m, addr_x, wr_en_x, clear_acc`) will need to be generated by a `control` module. Your control module will the contain control logic you require (e.g. counters, FSMs, etc.). Figure 7 shows the interconnections between your datapath and the control module.

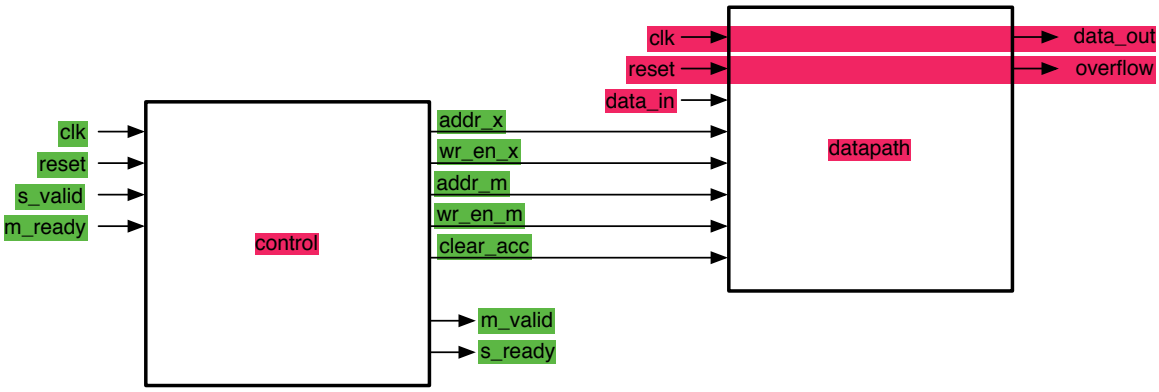

Figure 7. Interconnections between control unit and datapath.

For your top-level synthesizable module, use the following module name, port names, and port declarations:

```
module mvm3_part1(clk, reset, s_valid, m_ready, data_in, m_valid,
s_ready, data_out, overflow);
    input clk, reset, s_valid, m_ready;
    input signed [7:0] data_in;
    output signed [15:0] data_out;
    output m_valid, s_ready, overflow;
```

If necessary for your system, you can replace the last lines above with:

```
    output signed logic [15:0] data_out;
    output logic m_valid, s_ready, overflow;
```

as needed.

Example Testbench. To help you get started, we will provide you with a basic simple testbench. This testbench will demonstrate the desired input/output timing. To help simulate the fact that s_valid and m_ready can change at any time, this testbench uses a random number generator to set these values. On every cycle, it will randomly decide whether or not s_valid and/or m_ready are set. This randomization makes the test much more robust.

To take advantage of the randomization, you may want to run this testbench multiple times with different random seeds. You can set the random seed when you run vsim by typing:

```
vsim -sv_seed 42 -c check_timing
```

The number 42 is the seed. If you run this again with the seed set to 42, the random numbers will behave exactly the same way each time. So, when you re-run the testbench, if you change the number 42 to any other number, the random behavior will differ each time.

This testbench is available at:

```
/home/home4/pmilder/ese507/proj2/mvm3_part1_simple_tb.sv
```

Your tasks for Part 1 are:

1. Create this design in SystemVerilog. For ease of debugging and future extension, I suggest you use a top-level module (`mvm3_part1` as above), a control module, and a datapath module.

2. Write one or more testbenches to test your design ***well***. Include comments so I can understand how your tests work, and what the expected output should be. Use your testbench(es) to simulate your design. Part of the points for Part 1 will be related to the quality of the testbench. Don't forget to ensure that your testbenches simulate the fact that s_valid or m_ready may go low at any time.

3. Use Synopsys DesignCompiler to synthesize your design and find the maximum possible clock frequency. Save the resulting synthesis log file with a descriptive name. From this log, determine the design's area, power, maximum clock frequency, and the critical path's location in your logic.

4. In your report, include the following:

    a. How many arithmetic operations are required to multiply a 3×3 matrix with a vector of length 3? Then, generalize this: how many arithmetic operations are required to multiply a $k \times k$ with a vector of length k?

    b. Explain how your control module works. What are the steps it goes through? How does it keep track of its place in execution?

    c. Explain your verification strategy, and how your testbench works. How did you handle the s_valid and m_ready inputs?

    d. Report the area, power, frequency, and critical path location you determined from your synthesis report.

    e. From your simulation (and your understanding of your design), determine how many clock cycles the system takes to load one set of inputs, compute one matrix-vector multiplication of size k=3, and output the result, assuming that the testbench does not stall your design (that is, the testbench ensures that s_valid and m_ready are always asserted). Count from the first cycle of loading the input until the last cycle of outputting the result. Then, multiply this by the clock period to find the delay of the system (in nanoseconds).

    f. 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.

    g. Based on the synthesis power estimate, how much energy is consumed by your system while computing one matrix-vector multiplication (as measured in part e.)?

    h. We can define the *arithmetic operation count* as the number of additions and multiplications required to perform one matrix-vector multiplication. What is the operation count for your system? How much energy does your system consume *per arithmetic operation*?

Part 2: 3x3 Matrix-Vector Multiplier plus Vector Addition

Now, we will include the vector addition shown in equation (2) above. Now, your system will take its inputs in the following order: matrix M, then vector b, then vector x. So, when k=3, your first 9 inputs will be the matrix M, then the next three will be the vector b, and the

final three will be vector *x*. For simplicity, we will assume that the *b* vector is an 8-bit unsigned value. (That is, it is an 8-bit positive value.)

We will now need a memory to buffer the vector *b*. However, if we do this correctly, we can include these additions at no extra arithmetic cost. Consider the datapath shown in Figure 8.

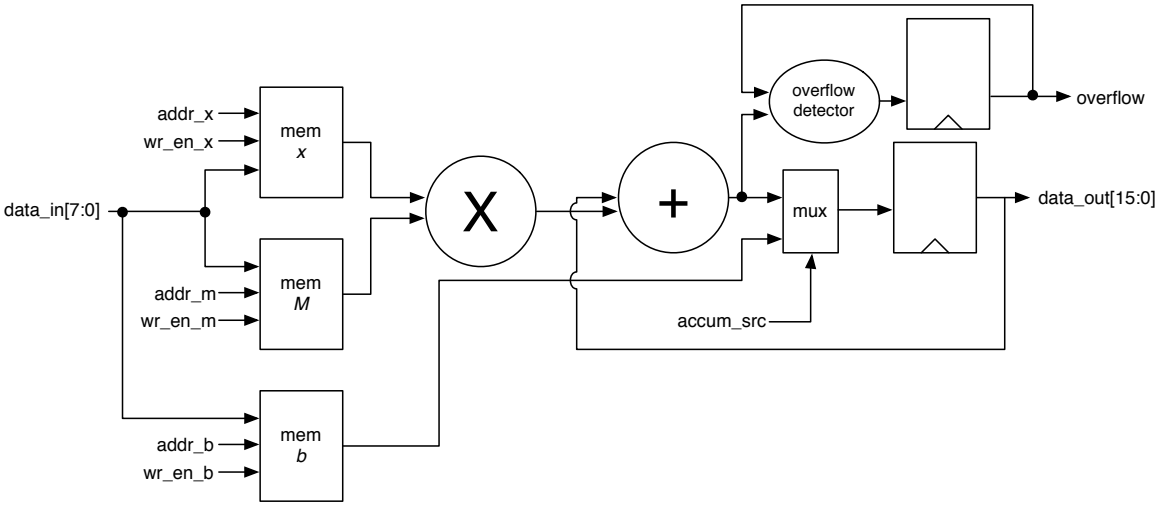

Figure 8. Datapath for Part 2.

Now, when we want to add in a *b* value, we can simply initialize the accumulation register to this value, rather than clearing the accumulator. That is, rather than resetting the accumulation register to 0, we will reset it to the appropriate value from the *b* memory.

Your tasks for part 2 are:

1. Create this design. Use the same top-level module inputs and outputs as from part 1, but change the name to `mvma3_part2`. Please note the prefix is "mvma" with the "a" corresponding to the vector addition we have added for this part.

2. Update your testbench from part 1 to test this design.

3. Again use Synopsys DesignCompiler to synthesize your design and find the maximum possible clock frequency. Save the resulting synthesis log file with a descriptive name. From this log, determine the design's area, power, maximum clock frequency, and the critical path's location in your logic.

In your report, include the following:

a. How many arithmetic operations do you now have for a *k* by *k* input matrix?

b. How does your control module change in part 2?

c. Repeat parts d–h from Part 1 #4 above for this new system.

Part 3: 4×4 Matrix-Vector Multiplier plus Vector Addition

In Part 3, you will adapt your design from Part 2 to make k=4. That is, your input matrix is now 4×4, and your vectors b, x and y have length 4. Then you will reason about how the design will change as k increases.

Use module name `mvma4_part3` for the top-level module of this design. Create this design, update your testbench, simulate your design, and synthesize it. In your report explain how the design changed. How difficult was it to change your control module?

If you wanted to build a design for much larger values of k, how would you do so? Would it be easy or difficult to change? In your report, explain exactly how your design would change as k increases. Be specific. Are there practical limits on the value of k? If so, what are they?

Again repeat steps d–h from Part 1 for your new design, including your responses in your report.

Part 4: Delay-Optimized System

Now, you will take your design from Part 3 and modify it to be as fast as possible, while keeping the input/output ports and timing as specified in the previous sections. Name your top-level synthesizable module `mvma4_part4`. You may change the internals of your design in any way possible, but the input/output behavior must not deviate from this specification. Some ideas you may want to pursue:

- increasing the parallelism. That is, building more adders, multipliers, and memories so that you can perform more operations concurrently,
- pipelining,
- modifying your control logic so that you can overlap input and output data (i.e., while the system is outputting data on `data_out`, it can begin taking in new data on `data_in`),
- using DesignWare components (including the pipelined versions). Please note that this does add some complexity to the simulation process; see below.

You may use DesignWare components for adders and multipliers, but you may not use the DesignWare multiply-and-accumulate unit. If you choose to use any DesignWare components, you will need to include them in your simulation (see slides from end of Class 13, October 12), and your testbench must still work correctly on your final design.

Use module name `mvm4a_part4` for the top level of this design. Create this design, update your testbench, simulate your design, and synthesize it. In your report explain:

a. What did you do to boost the speed? How well did it work? If you tried multiple things, explain what they were and whether or not they helped.

b. Collect the information parts d–h from Part 1 for your new design, and include them in your report.

c. Your new design performs the same computation as your design in Part 3, but it should be faster, larger, and consume higher power. Compare its area-delay product and energy per arithmetic operation with your Part 3 design. In these metrics, is your faster/larger design better or worse than your previous design?

d. If instead of optimizing for delay, what if your goal was to optimize for the overall lowest energy-per-operation? How would you build a matrix-vector-multiplication system to minimize energy?

e. Because you are constrained by the number of input and output ports, the maximum speed of your design here is limited. What could you do to make a design even faster, if you were allowed to change the input/output timing and ports?

A portion of the points allotted for Part 4 will be based on the quality of your optimizations and the final speed of your correctly functioning design. We will measure speed in seconds (the number of cycles times the clock period). In order to account for designs that overlap input and output data (item #3 in the list of suggestions above), we will measure cycles as the number of clock cycles required between when the system takes in the first element of the input matrix, and when it is capable of taking in the first input element of the *next* input matrix. (This way of measuring speed is *throughput*-based, as opposed to latency based.)

Code and Report Submission

1. **Code**

   You will turn in a single **.zip, .tar, or .tgz** file to Blackboard. ***Do not use a different archive format (including .rar).*** This compressed file should hold all of the files from your project. Organize them into three directories: part1, part2, part3, and part4. Put a `readme` file in each directory that explains what each file is. Make it very clear how to find your design and testbench for each part. I will be testing your designs using my testbench, so it is very important that you stick to the specification closely.

   Do not turn in things like ModelSim "work" directories or gate-level Verilog produced by synthesis.

2. **Report**

   Your report should include the information requested above. Include your report in the electronic hand-in with your code **(as a PDF file only)**.

3. Electronic Hand-in Process

To hand in your code, go to Blackboard –> Assignments -> Project 2. There you can upload your .zip, .tar, or .tgz file. You only need to hand in once per group, but make sure both partners' names are clear in your code and report.