Stony Brook University

Project 2: Matrix-Vector Multiplication

Issued: 9/30/15; Due: 10/21/15, 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. In part 1, you will construct a small basic system that takes as input a matrix and a vector, and outputs their product. In part 2, you will expand it to work with a larger input size. Then, in part 3, you will parallelize it to make it faster (but larger).

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
- clearly labeled synthesis reports for each time you are asked to synthesize a design
- a report that answers specific questions asked throughout this assignment

I 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)
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 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.

Background

Matrix-Vector Multiplication

We first begin by reviewing matrix-vector multiplication. As an example, let A_3 represent a square 3×3 matrix, and let x represent a (column) vector of length 3. The product $y = A_3 x$ is defined as:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} a_0 & a_1 & a_2 \\ a_3 & a_4 & a_5 \\ a_6 & a_7 & a_8 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} a_0x_0 + a_1x_1 + a_2x_2 \\ a_3x_0 + a_4x_1 + a_5x_2 \\ a_6x_0 + a_7x_1 + a_8x_2 \end{bmatrix} \quad (1)$$

So, this system takes in 12 values (the 3×3 matrix A_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 A_k and a length k input vector x, the system would thus take in k^2+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.

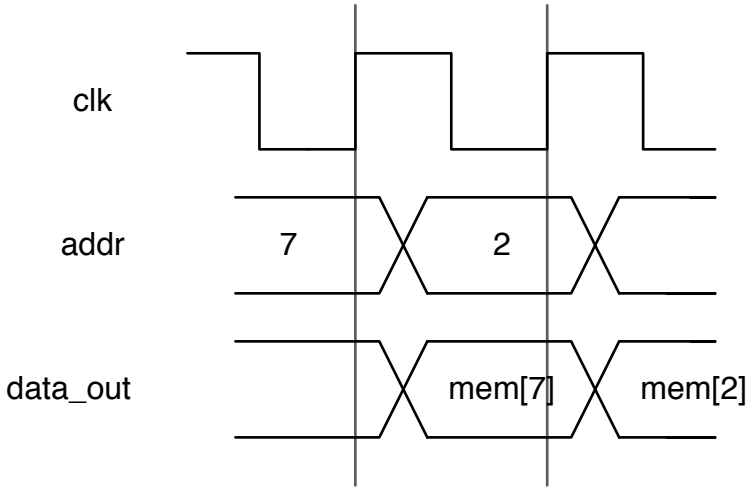

Figure 1. Timing of memory read.

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.

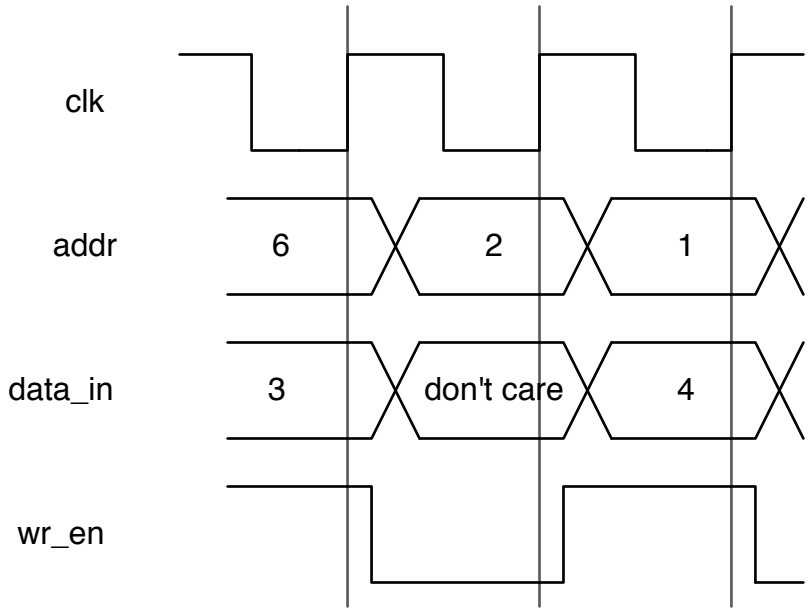

Figure 2. Timing of memory write.

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 cause a problem. 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.

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 a_0 and x_0 on the first cycle, a_1 and x_1 on the second cycle, and a_2 and x_2 on the third cycle.

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

Figure 3 shows the top-level view of your design. In addition to inputs `clk` and `reset`, the system takes a `start` signal used to control the system, and an 8-bit data input port. The system's outputs are a 16-bit data output port, and a 1-bit "done" signal, which the system will use to indicate that the calculation is complete (see below).

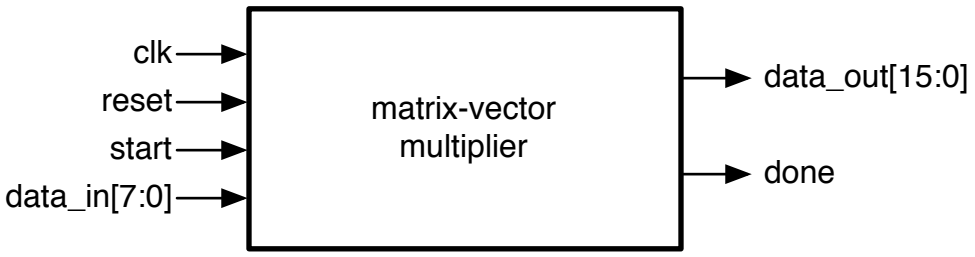

Figure 3. Top-level design

Input/Output Timing

In the following explanation of the system's input and output timing, we will continue to assume that your system multiplies a $k \times k$ input matrix A_k with a length k input vector x. However, it is important to understand that k is a *constant known at design time.* For example, in part 2, you will build a system where k=3. In other words, that hardware you build will be for a specific given value of k.

The system takes an input called `start`, which will be asserted one clock cycle *before* input data begins entering the system. Then, over the following k^2 cycles, the system will take as input the matrix A_k in row-order (a_0, a_1, a_2, a_3, …). Then, in the following k cycles, it will take the values of x. Figure 4 demonstrates the beginning of this process.

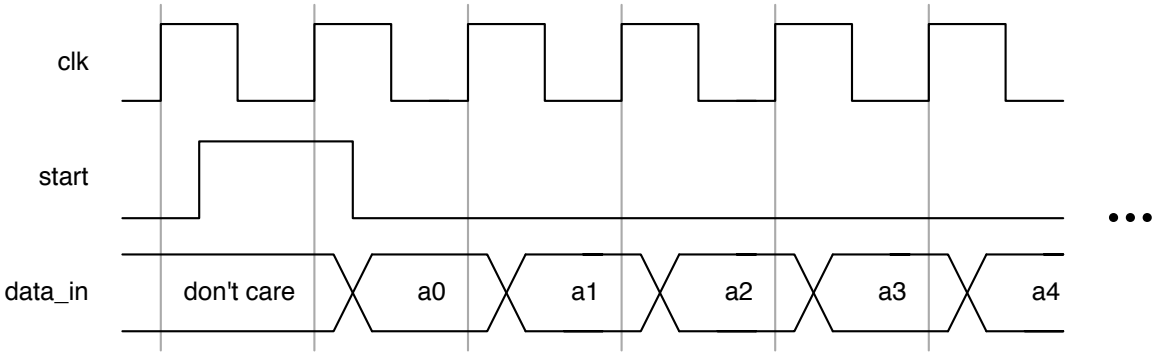

Figure 4. Input timing.

The system's output called `done` is analogous. The system should assert the `done` signal one clock cycle before it begins outputting the values of *y*. Then, on the following *k* cycles, the system will output the *k* values of *y*. Figure 5 demonstrates the beginning of this process.

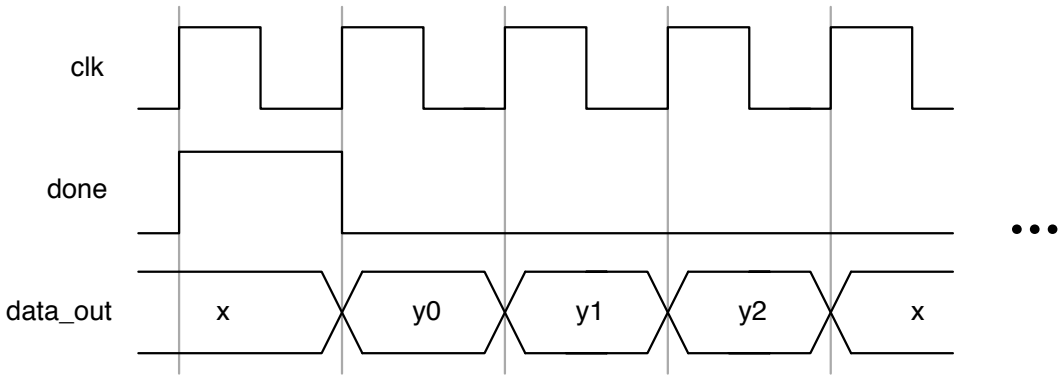

Figure 5. Output timing (assuming $k = 3$).

In Part 1 and Part 2, you will get started by building systems that we specify. Then, in Part 3, you will need to be creative to find ways to make your designs faster.

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.

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.)

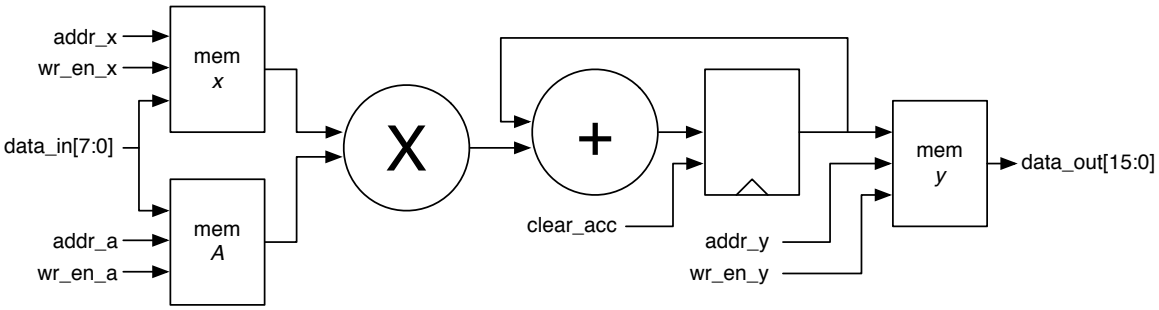

Figure 6. Datapath structure.

Note that many of the datapath's inputs are control signals. These signals (`addr_x`, `wr_en_x`, `addr_a`, `wr_en_a`, `clear_acc`, `addr_y`, `wr_en_y`) 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.

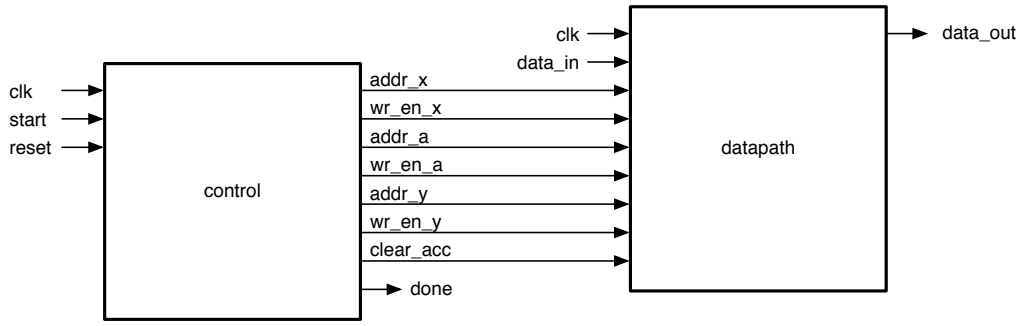

Figure 7. Interconnections between control unit and datapath.

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

```
module mvm3_part1(clk, reset, start, done, data_in, data_out);
    input clk, reset, start;
    output done;
    input signed [7:0] data_in;
    output signed [15:0] data_out;
```

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

```
    output signed logic [15:0] data_out;
```

Your tasks for Part 1 are:

1. Create this design in SystemVerilog. For ease of debugging and future extension, I suggest you use three modules: 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.

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.

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 compute one matrix-vector multiplication of size k=3, based on the number of clock cycles between when the `start` signal is given as input, and when the `done` output signal is asserted. 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.)

g. Based on the synthesis power estimate, how much energy is consumed by your system while computing one matrix-vector multiplication?

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: 4×4 Matrix-Vector Multiplier

In Part 2, you will adapt your design from Part 1 to make k=4. That is, your input matrix is now 4×4, and your input and output vectors x and y have length 4. Then you will reason about how the design will change as k increases.

Use module name `mvm4_part2` 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.

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

Part 3: Delay-Optimized 4×4 Matrix-Vector Multiplier

Now, you will take your design from Part 2 and modify it to be as fast as possible, while keeping the input/output ports and timing as detailed above (Figures 3, 4, and 5). Name your top-level synthesizable module `mvm4_part3`. You may change the internals of 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 Class 12, October 7), and your testbench must still work correctly on your final design.

Use module name `mvm4_part3` 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 2, but it should be faster, larger, and consume higher power. Compare its area-delay product and energy per arithmetic operation with your Part 2 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 3 will be based on 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, and part3. 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. You may include your report in the electronic hand-in with your code (as a PDF file only), or you may turn it in as a hard copy in class.

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.