Stony Brook University

Project 3: Hardware Generation Tool

Issued: 11/2/15; Milestone due: 11/16/15 11:59PM
Final project due: 12/4/15 5:00PM

1. Introduction

In Project 1, you studied the design, verification, and synthesis of a datapath that performed multiply-and-accumulate. Then, in Project 2 you combined this system with memories and a control module to construct a few varieties of matrix-vector multiplication (MVM) systems.

The goal of this project is to extend these ideas and create a piece of software that flexibly generates hardware implementations of MVM. Your software will take as input some parameters that describe the matrix and vector size, the input data precision, the desired parallelism, and the amount of pipelining. Your software will then generate the corresponding design and an appropriate testbench in SystemVerilog.

Your software can be written using any programming language you like, as long as the generator can be correctly compiled and run using the graduate CAD lab computers. I will provide you with a basic framework in C to get you started (see below). It is your responsibility to ensure that your code can run correctly on the CAD lab machines.

You will use your generation tool to produce many different types of designs, and you will evaluate (through synthesis) the quality of the tradeoffs produced. Please note that a substantial portion of the effort of this project is expected to be in the evaluation of your designs (Section 6).

You will turn in:

- your documented software, including instructions on how to compile and run it
- several examples of generated code (see instructions in Section 6)
- clearly labeled problem-free synthesis reports for each time you are asked to synthesize a design
- a report that answers the specific questions asked throughout this assignment

For your convenience, I have written an example piece of software (in C) that will show how to take in the required command-line parameters and output a file. It will also include instructions for how to compile and run it on our lab (Linux) computers. You can copy this code from `/home/home4/pmilder/ese507/proj3` on the lab computers (copy the whole directory—there are three files). It is up to you whether or not you use this as the basis for your project; I am providing it in case it helps.

As in past projects, 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 assignment
2. Carefully label and document your code
3. Organize and name your files your files as shown below.
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 quality of your generator and the designs it produces, 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.

2. New Input Control Signals

In Project 2, your system had one control input: `start`. Each time `start` was asserted, the system would then take as input a new $k \times k$ matrix A and a new k-element input vector x. This can be very wasteful if the user wants to use the same input matrix for multiple vectors. So, your **new design** will now take **three** control signals as input: `loadMatrix`, `loadVector`, and `start`. The basic idea is that the load signals are used to indicate you are loading a new matrix or vector, and the start signal is used to indicate that you would like the system to perform a computation based on the currently loaded matrix and vector.

When the `loadMatrix` signal is asserted on a positive clock edge, the system will take in a new matrix on the next k^2 clock cycles. When the `loadVector` signal is asserted on a positive clock edge, the system will take in a new vector on the next k clock cycles. Obviously, the system cannot do both of these things at once, because both the matrix and vector input share a single `data_in` signal. Finally, when the start signal is asserted on a positive clock edge, the system will begin computing the matrix-vector product based on whichever vector and matrix are stored in the input memories. [1]

The `done` output signal will function in the same way as in Project 2. When your computation is finished, `done` will be asserted for one cycle. Then on the following k cycles, the output vector y will be produced. Figure 1 shows the new top-level design for your matrix-vector multiplier systems.

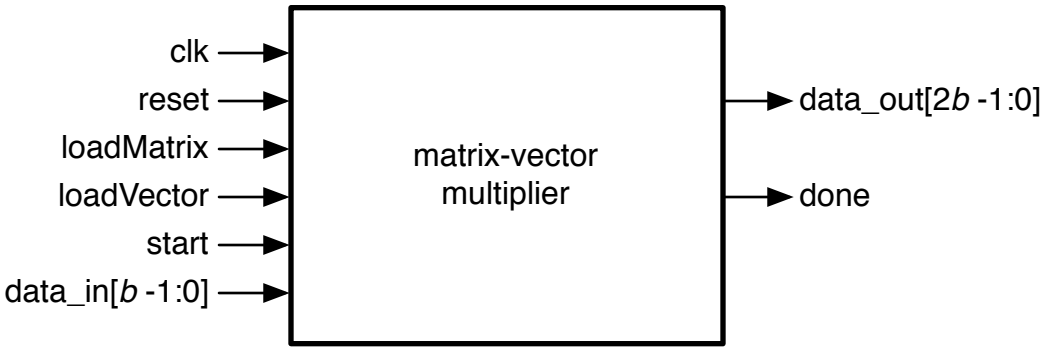

Figure 1. Top-level design.

[1] If this is the first action taken after "powering up" your circuit (that is, if your testbench asserts `start` without first storing valid data with `startMatrix` and `startVector`), then there will be no valid data stored in memory. Do not worry about this case. (If you simulate this, your system will simply output `X`. This is fine.)

3. Design Parameters

Now we describe a set of parameters that define a family of matrix-vector multiplication designs. These parameters will be inputs to your hardware generation tool.

Input Precision b (Number of Bits)

In Project 2, your input data words were 8-bit signed values. After multiplication, this doubled to 16 bits. For this project, we will use the parameter b to represent the number of bits per input word, so the system's outputs will be $2b$ bits each. Parameter b can be any integer ≥ 4.

Matrix and Vector Dimension k

Just as before, we will let k represent the dimension of our input vector x, and let our input matrix A be a $k \times k$ matrix. We will constrain k to be any integer ≥ 4.

Parallelism p

Our design space will use the parameter p to represent the degree of parallelism. We will allow two amounts of parallelism. First, when $p = 1$, we will build a single multiply-and-accumulate unit. (This is the baseline case, as in Project 2.) Second, when $p = k$, your system will contain k parallel multiply-and-accumulate units. For this parallel $p = k$ version, think carefully about the number of memories needed.

Pipelining g

Our design space will allow two pipelining options, which we will represent with the parameter g. When g=0, the datapath is unpipelined, and when g=1, the system will have one pipeline stage inserted into its datapath. This is shown in Figure 2, which demonstrates the datapath structure with and without the pipeline stage.

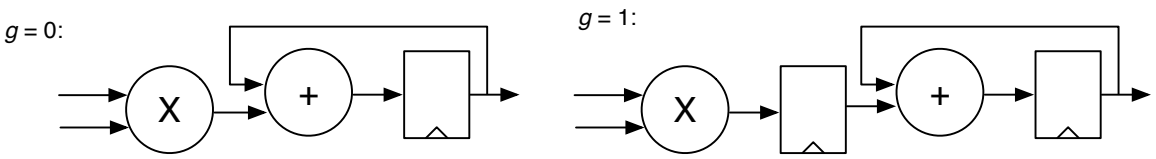

Figure 2. Left: Without pipelining (g=0). Right: Pipelined (g=1).

Design Space Summary

These four parameters k, p, b, and g define a space of possible designs for your MVM unit. You can represent a set of choices of these four parameters as the 4-tuple (k, p, b, g). So, (4, 1, 4, 0) will represent the *smallest* design in this design space (4×4 matrix, minimally parallel, 4 bits, unpipelined). By setting values for these four parameters, you are controlling tradeoffs between the *problem size*, *precision, area,* and *throughput* of your system.

4. Design Space and Hardware Generator

Based on the design parameters presented in Section 3, you will create a hardware generation tool. This is a piece of software that takes as input the parameters (*k, p, b, g*) and outputs a SystemVerilog implementation of the corresponding MVM design. Your generator should also produce a testbench that that you will use to verify the correctness of the generated design. Figure 3 demonstrates the input and output of your hardware generator.

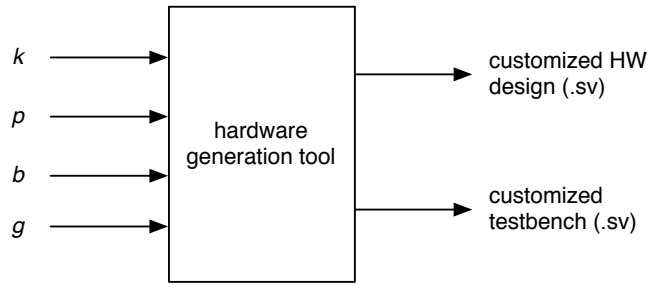

Figure 3. Flexible MVM Generator.

Note that we have **not** defined maximum values for *b* or *k*. Is it possible to support infinitely high values for these parameters? Think carefully about how you will deal with this in your generator. What are the practical limits, and why?

Naming Conventions

When you generate a design with dimension *k*, parallelism *p*, bitwidth *b*, and pipelining g, use following file naming convention:

```
p_milder_mvm_k_p_b_g.sv
```

Where you will replace "`p_milder`" with your first initial and last name,[2] **and you will replace `k`, `p`, `b`, and `g` with the values of the parameters for the design generated**.

Similarly, for your top-level synthesizable module, use the following module template (where again you will replace `k`, `p`, `b`, and `g` with the appropriate parameter values):

```
module mvm_k_p_b_g(clk, reset, loadMatrix, loadVector, start,
done, data_in, data_out);
    input clk, reset, loadMatrix, loadVector, start;
    output done;
    input signed [b-1:0] data_in;
    output signed [2*b-1:0] data_out;
```

Testbenches

By constructing a hardware generator, you will be able to automatically produce a very wide space of designs. However, this can lead to another problem: how do you know that the designs you produce are correct?

[2] If you are working with a partner, please just use one partner's name—preferably whichever partner will submit the code on Blackboard.

To help with this problem your generator will also produce a customized testbench for the given parameters. Think carefully about how to construct this testbench and how you will use it to verify the correctness of the generated code. Similar to before, use the following naming convention for your testbench (with appropriate changes of name and parameters):

```
p_milder_mvm_tb_k_p_b_g.sv
```

Generator Behavior

Your generator should be executable entirely from the command line on one of our lab machines. The generator must take four parameters at the command line: *k, p, b,* and *g* in the following way. For example, if (*k, p, b, g*) = (2, 4, 12, 0):

```
./gen 2 4 12 0
```

Then your generator must produce the output files described above. To get you started, I have provided a starting point using the C language at:

`/home/home4/pmilder/ese507/proj3`.

You may use any other programming language you like, but you must ensure that its inputs are provided in the same way as this example.

5. Milestone

The milestone is an intermediate set of tasks, which will be due slightly less than halfway through the project (11/16). There are two purposes to the milestone: (1) to help you pace your progress so you do not fall behind, and (2) to allow you to get feedback and correct any problems before the final submission.

For the milestone, you must produce a simplified hardware generation tool that has some flexibility. Your milestone design will take in parameters *b* and *k* (while assuming that *p*=1 and *g*=0), and will produce the corresponding design and testbench.

For the milestone, turn in the following:

- your generator code (software) so far
- four designs and testbenches generated with the following parameters (*p*=1, *g*=0):
    - *b*=6, *k*=4
    - *b*=11, *k*=5
    - *b*=16, *k*=8
    - *b*=8, *k*=32
- four synthesis reports (one for each design). Make sure there are no errors or major warnings (missing files, inferred latches, etc.).
- a very short text file `report.txt` that gives your name(s) and explains whether or not you fully completed the milestone. If you have completed all milestone tasks successfully, please just write "All tasks successful." If you have not completed them all successfully, explain what you were not able to finish.

Create a .zip, .tar, or .tgz archive with the files listed above, and submit it on Blackboard (under "Project 3 Milestone").

6. Evaluation and Report

After completing your full generator and simulating the results to verify the correctness of your generated designs, you will answer some specific questions about your work, and you will use Synopsys DesignCompiler to evaluate some of the designs produced by your generator.

In your report, address each of the following:

1. In hardware generators, scalability and flexibility can be difficult. In your report, explain how you made your generator capable of handling the flexibility required by the parameters (*k, p, b, g*). Were any of these particularly easy or difficult to support? How did you handle the fact that there were no maximum defined values on *k* and *b*? Are there practical limits on these values? If so, what are they, and why?

2. Describe how you designed your control module/FSM. How did you structure the design so that it can be changed as the *k* parameter changes? Explain how the structure changes as *k* grows.

3. Describe how you implemented the parallel (*p*=*k*) designs. Explain your structure. What extra logic and storage elements did you need to add? Did you find any clever optimizations to reduce cost?

4. Explain your testbench strategy. How do your testbenches work? Justify why your testbenches test the designs sufficiently. How difficult was it to incorporate the flexibility of the various parameters into the testbench itself?

5. In Section 3, we discussed how the parameters (*k, p, b, g*) allow various tradeoffs between *problem size, costs, precision*, and *performance*. Explain how these tradeoffs work at a conceptual level. In other words, explain how you would expect changing each parameter to affect these metrics. Be specific.

Now, you will use your generator to produce several designs, synthesize them, and evaluate their cost and performance as the parameters change. For each of the designs you will be synthesizing below, turn in your `.sv` file and the Synopsys output log file (use the same naming convention but with extension `.txt`). Each time you synthesize, aim for the highest reachable clock frequency.

As we have previously discussed, it is very important to carefully understand your synthesis reports. If there are *any* errors listed at all, then it means your entire design did not synthesize correctly. This can be caused by things like missing files or typos as well as serious design problems. If any error is shown, you *must* correct it and re-synthesize. Warnings can also be problematic. Some ("signed to unsigned conversion") may not matter, but others ("inferred latches" or "unresolved references") are very big problems. If your synthesis report shows either of these, make sure to correct the problem and re-synthesize.

6. Now, we will use synthesis to evaluate how the area and power of an implementation scale as the input precision *b* changes. Use your generator to produce four designs with *b*=8, 12, 16, and 20, while you keep *k*=8, *p*=8, and *g*=1.

Then produce two graphs that illustrate: (1) power versus b and (2) area versus b for these designs. Describe where the critical path is located in each design.

7. Next, we will evaluate how throughput, area, and power scale as k changes. Use your generator to produce four designs with k=4, 8, 16, and 32, while you keep p=1, b=8, and g=1. Synthesize each design, and graph: (1) power versus k, (2) area versus k, and (3) throughput versus k. Does the location of the critical path change as k changes?

   Throughput is defined as the number of data inputs processed per second. To calculate this, you will first determine the average number of data words per cycle, and multiply this by the clock frequency f.

   We will calculate the *words per cycle* assuming that our system keeps a fixed matrix A. (That is, assume that a matrix is stored in memory and that we will use `loadVector` followed by `start` for every input.) Under these assumptions, for every MVM computed, your system processes k input words. How many cycles does this computation take? Let c represent the number of cycles needed to process these k input words. Then, your throughput will be (k/c) words per cycle times f cycles per second. In your report, include the values of c that you found for each design, and explain how you found them.

8. In the previous step, you used pipelined designs (g=1). How do the costs and performance change if you use unpipelined designs? Generate and synthesize designs without pipelining using parameters (k, p, b, g) = (4, 1, 8, 0) and (32, 1, 8, 0) and compare these two designs to their pipelined counterparts from question 7. How does the critical path change?

9. Lastly, you need to evaluate how the designs change when you increase parallelism (by setting p=k). Now, generate and synthesize four parallel designs: k=4, 8, 16, and 32, with p=k, b=8, and g=1. Graph: (1) power versus k, (2) area versus k, and (3) throughput versus k.

   These parallel designs will be faster but more expensive than their counterparts from question 7. Which are more *efficient:* the more-parallel p=k designs, or the less-parallel p=1 designs? Justify your answer quantitatively.

7. Final Submission

You will turn in a single **.zip, .tar,** or **.tgz** file to Blackboard. This compressed file should hold all of the files from your project. Your submission directory should have three sub-directories: `src/`, `hdl/`, and `report/`.

The `src/` directory will hold your generation software. Please make sure you include information about how to compile it. (`Makefile`s are helpful; `README` files are essential.)

The `hdl/` directory should hold the SystemVerilog files you generated to complete the evaluation (specified in Section 7 of the assignment). Also include your testbenches and synthesis output files here. Please use the naming conventions given above.

The `report/` directory will hold your report (PDF format only). Your report should answer all of the questions above.

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

To hand in your code, go to Blackboard –> Assignments -> Project 3. There you can upload your .zip, .tar, or .tgz file. If you are working in a group of two, please only submit the assignment under one partner's Blackboard account (it doesn't matter which).

Your final upload is due on Friday, December 4, at 5:00PM. No extensions to this deadline will be possible.