Stony Brook University

Project 3: Hardware Generation Tool

Issued: 11/6/17; Milestone due: 11/20/17 11:59PM
Final project due: 12/8/17 11:59PM

Partner

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

- Send an email to peter.milder@stonybrook.edu with the subject "ESE 507 Project 3 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 3. (If you want to change for later projects, you may.)

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, including the addition of an output vector.

The goal of this project is to extend these ideas and create a piece of software that flexibly generates hardware to accelerate the evaluation of layers of artificial neural networks.

In Part 1, your software will take as input some parameters that describe the matrix dimensions, the input data precision, and the desired parallelism. Your software will then generate the corresponding design in SystemVerilog. Part 2 adds parallelism.

Then Part 3 will extend this idea to generate a piece of hardware for a multi-layer network, where three layers generated using your solution to Part 2 are composed together to form a larger system. In Part 3, your system will take as input a multiplier budget (i.e., the maximum number of multipliers your design may use) as well as the dimensions for *each* of the three layers. Your generator will use this information to choose *how parallel* to make each of the layers.

You are being provided with a software framework written in C++ that will help you get started. This framework additionally demonstrates some key functionality you will need to use to complete the assignment. This can be found in the directory `/home/home4/pmilder/ese507/proj3` (copy the whole directory—there are several files). You may choose to start from this framework or to build your own in another language (as long as your solution can be correctly compiled and run using the graduate CAD lab computers). It is your responsibility to make sure your work functions correctly on the CAD lab computers.

In Projects 1 and 2, a substantial part of the problem was to generate good testbenches. For this project, we are providing you with a testbench generator and test scripts that will check your work. This is also included in the directory mentioned above.

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 8).

You will turn in:

- your documented software, including instructions on how to compile and run it
- several examples of generated designs (see instructions in Section 8)
- clearly labeled problem-free synthesis reports for each time you are asked to synthesize a design
- a report that answers the specific questions asked in Section 8

To help us evaluate your work it is very important to:

1. Make sure your systems match all problem specifications
2. Carefully label and document your code
3. Organize and name 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, 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 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.

Getting Started

Start by reading this entire document to give you a general sense of the scope of the project. Section 2 gives an overview of neural networks and details the problem you will be solving. Section 3 discusses ways that you will be extending your work from Project 2. Sections 4–6 describe Part 1, 2, and 3 of the project. Briefly, the goal of part 1 is to adapt your design from Project 2 into a generator for basic neural network layers. Part 2 adds parallelism to the design. Lastly, in Part 3 you will build a generator that connects three neural network layers together, and aims to optimize the speed of the system given a limited number of multipliers.

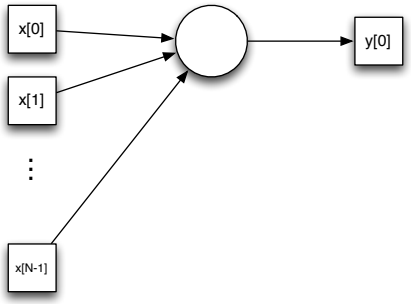
2. Neural Networks

Artificial neural networks are a biologically inspired way for a computer to "learn" based on observed data. Neural networks are nearly ubiquitous in modern machine learning, forming the basis for many problems such as image classification, natural language processing, fraud detection, and many other problems. This section is intended to give a very brief overview

of this area. A good reference to learn more is "Neural Networks and Deep Learning," by Michael Nielsen, a free e-book available online.[1]

A neural network is a computational model inspired by the brain. In its simplest form, each "neuron" takes in a number of signals (e.g. from other neurons or triggered from external stimuli), and performs a weighted sum of its inputs plus a constant, followed by a nonlinear function $f(\,)$.

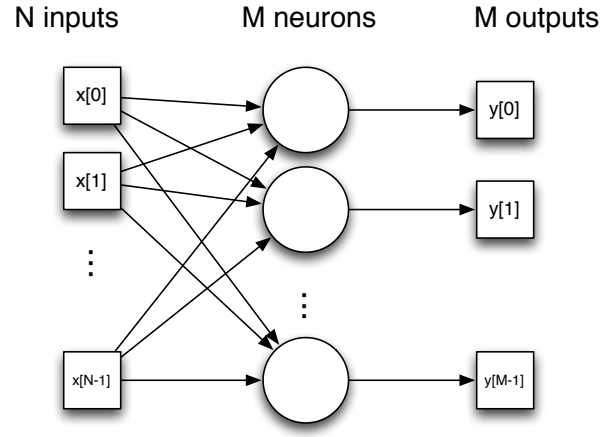
Below we see a neuron that takes inputs x[0], ..., x[N-1]. Each input is multiplied by a "weight" w[n], and summed with a constant value b (called the "bias"). Lastly, our nonlinear f function simply takes the maximum value of that result and 0. (This is called a rectified linear unit, or "ReLU." Other non-linear functions, such as the "sigmoid" are used in other contexts, but we will only use ReLU in this project.)



$$= f\left(\sum_{n=0}^{N-1} x[n]w[n] + b\right)$$

$$f(x) = \max(x, 0)$$

A layer of neurons is a set where each neuron takes the same N inputs, but has a different set of weights and a different bias value b:



Here, each output *y[m]* is computed as:

$$y[m] = f\left(\sum_{n=0}^{N-1} W[m][n] \cdot x[n] + b[m]\right)$$

where W is an M by N matrix, with each of the M rows holding the N weights for one neuron.

[1] http://neuralnetworksanddeeplearning.com

We can write this operation simply as a matrix-vector multiplication plus a vector addition—like project 2—with the extra inclusion of the *f* function:

$$y = f(Wx + b) \tag{1}$$

where:

- *W* is an *M* by *N* matrix, representing the "weights" of the neural layer
- *x* is a column-vector of length *N*, representing the inputs
- *b* is a column-vector of length *M*, representing the bias value of each neuron
- *f* ( ) applies a point-wise ReLU function: any negative value in the vector is replaced with 0, and
- *y* is a column-vector of length *M*, representing the outputs.

Training versus Evaluation

The process described above is the "evaluation" of a neural network layer—computing the layer's outputs based on its inputs, given the set of weights and bias values. In order to do so, one must determine what those values are for a specific application through a process called "learning." Typically, this is done using a set of training data (inputs whose "correct" outputs are already known) and performing an operation called stochastic gradient descent that aims to determine the correct parameter values based on the observed data.

In our project, we will be building systems to efficiently evaluate neural network layers whose parameters are already known, so we will not consider training.

3. Basic Changes from Project 2

In Project 2 Parts 2–4, you built systems that multiplied a vector *x* with a matrix (which we will now call *W* because it represents the weights of the neural network layer) and added a vector *b*. You will first be making a number of modifications to this system:

1. **Constant values for *W* and *b***: In Project 2, you took the *W* matrix and *b* vector as inputs to the system. In Project 3, we will instead be treating these values as *constants* that are chosen at the time your hardware generation program is run. This means you can store those values in ROMs—the handout code gives examples of how to do this.

2. **Non-square matrix:** In Project 2, we assumed that the *W* matrix was square (you built 3x3 and 4x4 versions). In this project, this matrix can be rectangular: M by N as in the example above. This means your system's input and output vectors can have *different sizes*. (The input vector *x* has length *N* while the output vector *y* has length *M*.)

3. **ReLU function *f***: as described in Section 2, we will be using a Rectified Linear Unit (ReLU) function on the outputs of your system. This simply means that after you calculate a final result in the accumulator, your system needs to check if it is negative; if it is, it simply replaces it with 0.

4. **Bitwidth:** In Project 2, we assumed all inputs were 8 bit values and that the outputs are 16 bits. Here, for simplicity your system will use the same number of bits for all inputs and outputs. This value will be a parameter.

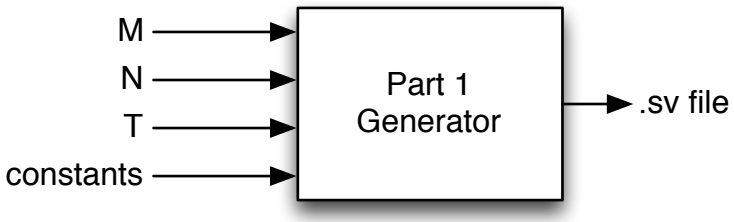
4. Part 1: Basic Generator for One Neural Network Layer

The goal of Part 1 is to build a hardware generator that produces a SystemVerilog design for one neural network layer (equation (1) above), controllable by a number of parameters. These parameters are:

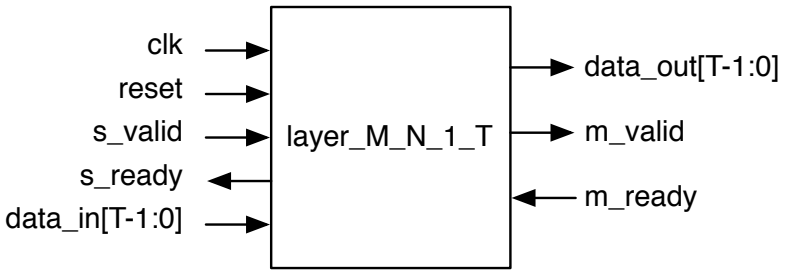
1. **Matrix dimensions *M* and *N*.** As described above, our matrix *W* will have *M* rows and *N* columns. This also means that our input vector *x* will have *N* elements, and the *y* and *b* vectors will each have *M* elements.

2. **Bit-width *T*.** The parameter *T* represents the number of bits used as a datatype everywhere in your system. Assume that $4 \le T \le 32$. (Note that unlike in Project 2, we will use the same number of bits everywhere, for simplicity.) Obviously, overflow is possible; ***for this project we will not worry about detecting overflow***. (Although it would be easy to include logic to detect when overflow exists like in earlier projects, or to use *saturating* arithmetic).

3. **Values for the *W* matrix and *b* vector.** Your generator will read in a text file that contains the constant values your system should use for *W* and *b*. This text file contains one integer (in base 10) per line. The first *M*N* lines give the values of W (in row-major order). Then, the next *M* lines give the *M* values of *b*. This part may sound complicated, but it is actually very easy to deal with. The handout code contains an example of how to read this file into a vector in C++, and then how to use those values to produce ROMs.



The hardware your generator produces should use the same input/output ports and protocol as in Project 2, except now we will ignore overflow, and the system does not need to take *W* or *b* as inputs; its only input data are vector *x* and its only output data are the elements of vector *y*.



For part 1, make your module's top-level module have name `layer_M_N_1_T`, where the *M*, *N*, and *T* parameters are replaced with their actual parameter values, e.g., `layer_4_5_1_16`. Use the following port declarations, e.g., if *M*=4, *N*=5, and *T*=16:

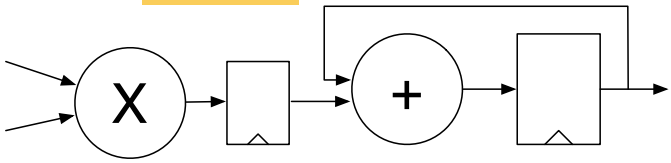
```
module layer_4_5_1_16(clk, reset, s_valid, m_ready, data_in,
m_valid, s_ready, data_out);
    parameter T=16;
    input clk, reset, s_valid, m_ready;
    input signed [T-1:0] data_in;
    output signed [T-1:0] data_out;
    output m_valid, s_ready;
```

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

```
    output signed logic [T-1:0] data_out;
    output logic m_valid, s_ready;
```

as needed.

In Project 2, many of you pipelined your design by adding a register between the multiplier and adder. You should do this here as well:



Input and Output

Your system will use the exact same input/output protocol and signaling as in Project 2. However, there is one key difference here: now your system's input will only be the vector *x*; it no longer needs to take matrix *W* or vector *b* as input because these are stored in ROMs. So, your system's input data will simply be an *N*-element input vector *x*, and its only output data will be the *M*-element output vector *y*.

Generator Behavior

Your generator should be executable entirely from the command line on one of our lab machines. The generator must take several parameters at the command line. Note that the provided reference code is already set-up to handle these parameters for you.

The first parameter specifies the mode. For parts 1 and 2 of the project, use mode 1. In mode 1, your generator will take as input the following parameters in the following order:

```
./gen 1 M N 1 T const_file
```

Where `M`, `N`, and `T` are the three parameters described above, and `const_file` is the name of the file that stores your layer's *W* and *b* constants. (Note that the first parameter value 1 indicates we are running in mode 1. The second parameter 1 listed above is for a new parameter that you will add in Part 2, described below.)

The generator will store the result in a file whose name matches the top-level module: `layer_M_N_1_T.sv`, where the parameters M, N, and T are replaced with the actual parameter values.

For example, if you want to implement a layer with a 4 by 5 matrix, using 16 bits, you have constants stored in a file called `const.txt`, you would run:

```
./gen 1 4 5 1 16 const.txt
```

The result would be stored in `layer_4_5_1_16.sv`.

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, and that we are able to run your generator correctly on the CAD lab Linux computers.

Our code for getting started is located at:

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

Copy this *entire directory* into your work directory. To compile the program, simply type:

```
make
```

Then you can run the program as described above.

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? To help you, we provide you with a testbench generator you can use to test your designs in all three parts of this, as well as test scripts that will use your generator to create a design, use the testbench generator to create the testbench, and run it for you.

The testbench code is also located in directory:

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

The testbench generator code will also be compiled when you run

```
make
```

The testbench generator produces a .sv testbench file as well as three data files that specify the input values to test, the expected output values, and the constants for *W* and *b*.

To run the testbench generator, run

```
./testgen 1 M N 1 T
```

(where you replace M N and T with your values for those parameters.

This will produce four files:

- `tb_layer_M_N_1_T.sv` the testbench file
- `tb_layer_M_N_1_T.in` the inputs to test
- `tb_layer_M_N_1_T.exp` the expected results
- `const_M_N_1_T.txt` the constants to give your generator

Then, you would generate the accompanying code with:

```
./gen 1 M N 1 T const_M_N_1_T.txt
```

(where again M, N, P, and T are replaced with your parameter values). This will produce your `layer_M_N_1_T.sv` file.

Then, to simulate:

```
vlog layer_M_N_1_T.sv tb_layer_M_N_1_T.sv
vsim tb_layer_M_N_1_T
```

(where again M, N, and T are replaced with your parameter values).

The simulation will report any errors.

Test Script

To test everything easily, we have provided a test script that generates the testbench, generates the design, and simulates the design. Running this script will allow you to do all three with one command line.

For part 1, you can run:

```
./testmodeone M N 1 T
```

(where you will replace the M, N, and T with your parameters).

For example, if you want to implement and test a layer with a 4 by 5 matrix, using 16 bits, you would run:

```
./testmodeone 4 5 1 16
```

Getting Started

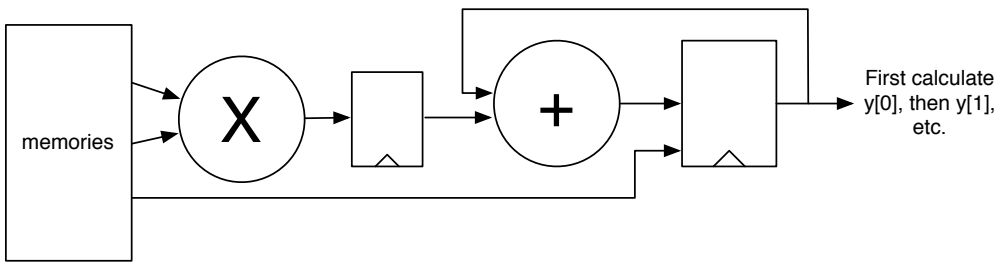
Start by copying our handout code to your work directory. (Please make sure you set the permissions correctly so your work is not visible to others.) Then, compile the code and try running it. Look at the output file that is produced. Look through `main.cc` and understand how the code is structured. The key to part 1 and part 2 is the `genLayer()` function. Read the comments. Also note how the system demonstrates how to create ROMs.

5. Part 2: Adding Parallelism to Your System

For this part, you will parallelize your design by increasing the number of multiply-accumulate units used in parallel. In Part 4 of Project 2, many of you considered ways to parallelize your system. Here, you will take a parameter that specifies the *amount* of parallelism in terms of the number of multiply-accumulate units you will use. There are several types of parallelism that can be used, but we will focus on only one.

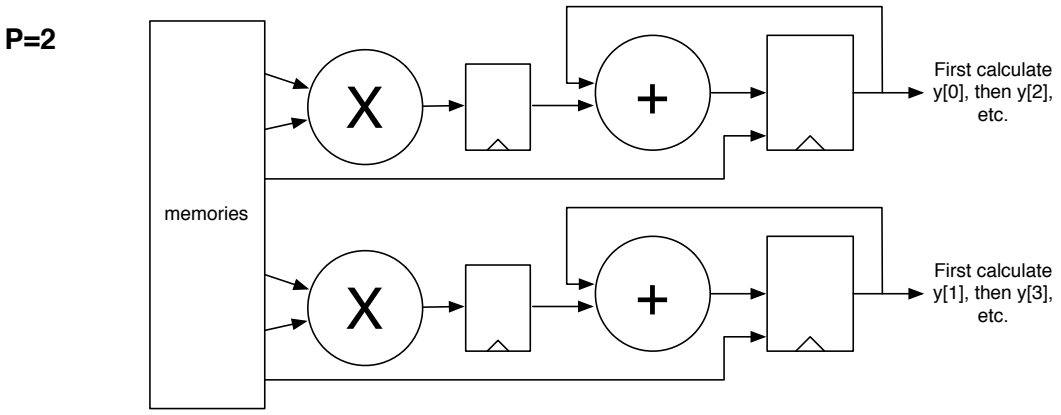
In Part 1, you used one multiply-accumulate unit, and you used it to compute one output value at a time:

P=1



First, the accumulator is used to compute y[0], then it is used to compute y[1], and so on. We will define the parameter *P* to represent the number of parallel multiply-accumulate units, so in the diagram above, *P*=1.

Now, as *P* increases, you will use *P* multiply-accumulate units to compute *P* outputs at the same time. E.g., for *P*=2:

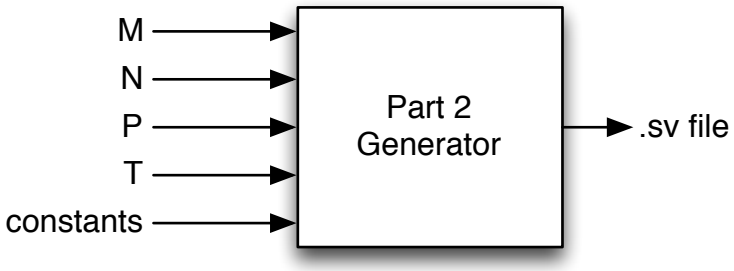


Now the first accumulator calculates y[0] while the second accumulator calculates y[1]. Then, they will compute y[2] and y[3] concurrently, and so on. You will need to reorganize how your system's memories are structured (since each multiplier-accumulate unit will need different constant values). You also will need to make sure the output logic takes into account the fact that you have multiple values being produced.

For Part 2, your task is to add the parameter *P* to your generator. For convenience, we will constrain *P* such that *M*/*P* is an integer. (Recall: *M* represents the number of output values your layer produces.) Therefore, if your system outputs *M*=16 words, then legal values of *P* are 1, 2, 4, 8, and 16. However if *M*=13, the only legal values of *P* are 1 and 13.

Generator and Testbench

For Part 2, you will still use mode=1 when running the generator. However, now you will be able to adjust the *P* parameter (which you kept to a constant 1 in Part 1).



You will run the generator for Part 2 using:

```
./gen 1 M N P T const_file
```

Note that changing the parallelism will change the *internal* structure of your neural network layer, but it does not change the top-level ports. Make your module's top-level module have name `layer_M_N_P_T`, where the *M*, *N*, *P*, and *T* parameters are replaced with their actual

parameter values, e.g., `layer_4_5_4_16`. Use the following ports declarations, e.g., if *M*=4, *N*=5, *P*=4, and *T*=16:

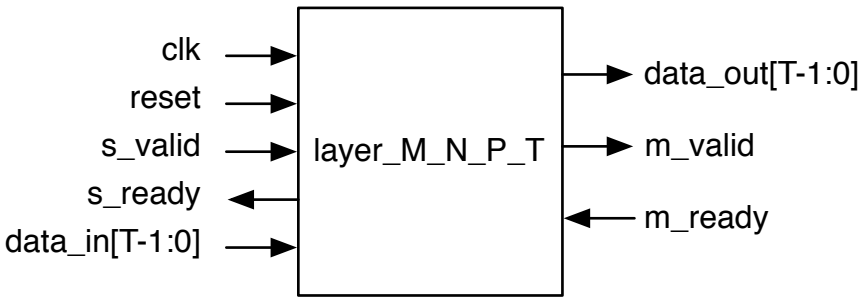
```
module layer_4_5_4_16(clk, reset, s_valid, m_ready, data_in,
m_valid, s_ready, data_out);
    parameter T=16;
    input clk, reset, s_valid, m_ready;
    input signed [T-1:0] data_in;
    output signed [T-1:0] data_out;
    output m_valid, s_ready;
```

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

```
    output signed logic [T-1:0] data_out;
    output logic m_valid, s_ready;
```

as needed.

Since the top-level input/output specification doesn't change from Part 1, you can use the same testbenches for Part 1 and Part 2.



For part 2, you can run the testbench generator with:

```
./testgen 1 M N P T
```

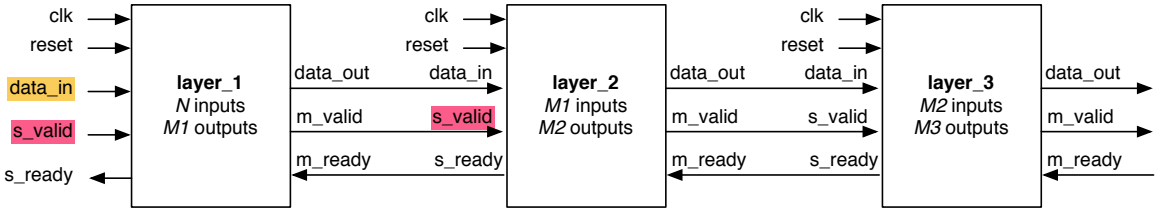
You can run the test script (which again will generate a testbench, generate a design, and simulate the design):

```
./testmodeone M N P T
```

where in both cases M, N, P, and T are replaced with the appropriate parameters (e.g., 4 5 4 16).

6. Part 3: Generating and Optimizing a Three-Layer Network

Neural networks typically consist of multiple layers. In this part of the project, you will extend your generator to produce a three-layer network.



Each of the three layers will be generated using your result from Part 2. Now, the output from Layer 1 will be directly connected to the input ports of layer 2, and so on.

Note that this creates a new constraint: the number of elements in the output vector of Layer 1 must be equal to the number of elements in the input vector of Layer 2. Therefore, we will define a new set of parameters to describe these layers.

We will say that Layer 1 has dimensions *M1* by *N* (therefore its output vector is length *M1*), then Layer 2 will have dimensions *M2* by *M1*, and Layer 3 will have dimensions *M3* by *M2*. So, Layer 1 will take an *N*-element vector and produce an *M1*-element output vector. Layer 2 takes an *M1*-element input vector and produces an *M2*-element output vector. Layer 3 takes an *M2*-element input vector and produces an *M3*-element output vector.

Additionally, we will allow the *P* parameters of the layers to be independent. That is, each layer can have a different amount of parallelism: *P1*, *P2*, and *P3*.

Optimization Problem

Notice that the number of clock cycles each layer takes will now depend on its dimensions and its parallelism. This leads to an opportunity for optimization: given a *multiplier budget L*, how do you choose the *P1*, *P2*, and *P3* parameters in order to maximize speed while using no more than *L* multiply-accumulate units? Necessarily, $L \geq 3$ (since there's no way for us to produce a layer with less than one multiply-accumulate unit). Similarly, $L \leq (M1+M2+M3)$, because this would enable the maximum parallelism for each layer.

Your optimization problem is: given *N*, *M1*, *M2*, *M3*, and *L*, determine the values of *P1*, *P2*, and *P3* that will maximize the throughput of your design.

Generator

For this task, we will add a new mode to your generator. When mode=2, we will be specifying the parameters for a three-layer network:

```
./gen 2 N M1 M2 M3 L T const_file
```

Your generator will determine the P1, P2, and P3 parameters. Then it will generate the three layers (using your result from Part 2). Lastly, it should generate a new top-level module that connects the three layers together.

Your module should be named `network_N_M1_M2_M3_L_T`, and the result should be stored in a file called `network_N_M1_M2_M3_L_T.sv`, where again the parameters N, M1, etc., are replaced with their actual values.

In Part 1 and Part 2, we used a text file to specify the constant values for the *W* matrix and *b* vector. Now, this file will need to store ***all*** the constants for all three layers. The first layer will require the first *N*M1+M1* elements; the second layer will use the next *M2*M1+M2* elements, and the last layer will use the final *M3*M2+M3* values from the constant file. We have provided you example code illustrating this in the `genAllLayers()` function of the handout code.

Testbench

We will similarly add a new mode to our testbench generator for part 3. You can now generate a testbench with: For part 2, you can run the testbench generator with:

```
./testgen 2 N M1 M2 M3 L T
```

You can run a new test script (which again will generate a testbench, generate a design, and simulate the design):

```
./testmodetwo N M1 M2 M3 L T
```

where in both cases M1, M2, M3, N, L, and T are replaced with the appropriate parameters. (Don't forget: L indicates the multiplier budget).

7. Milestone: Due 11/20/17

The milestone is an intermediate set of tasks, which will be due on 11/20. For the milestone, you are required to finish Part 1. This is a little more than halfway through the project's allocated time. ***Ideally, by this date you should also be finished with Part 2.***

For the milestone, turn in the following (to Blackboard):

- your generator code (software) so far
- four designs produced with the following parameters (P=1):
  - M=4, N=4, T=16
  - M=16, N=12, T=20
  - M=5, N=2, T=9
  - M=13, N=16, T=32

  and tested with the testbench we provide
- 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").

8. 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 (matrix sizes, parallelism, number of bits). Were any of these particularly easy or difficult to support? How did you handle the fact that there were

no maximum defined values on *M* and *N*? Are there practical limits on these values? Why or why not?

2. Describe how you designed your control module/FSM. How did you structure the design so that it can be changed as the *M* and *N* parameters change? Explain how the structure changes as these parameters grow.

3. Describe how you implemented the parallel (*P*>1) 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. Our design parameters *M*, *N*, *P*, and *T* allow various tradeoffs between:
   - problem size *(e.g., how big of a neural network layer we can compute)*
   - costs *(e.g., area, power, energy)*
   - precision (i.*e., how many bits of precision are used)*, and
   - performance (*e.g., throughput and latency)*.

   Explain how these tradeoffs work at a conceptual level. In other words, explain how changing the four parameters listed above affects these four metrics. **Be specific and think carefully.**

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. (For the log file, use the same base name as the `.sv` file but with extension `.txt`). Each time you synthesize, aim for the highest reachable clock frequency. Additionally, it is important that you use real constants in your ROMs when you synthesize. If you do not do this, e.g., if every constant is "0", the system will simplify your design dramatically (because it knows this ROM will always output 0, it will ignore it, and if the ROM always outputs 0, the multiplier's output is always 0, and so on). Every time you generate a design to synthesize, set its constant parameter file to be one generated by our random testbench generator.

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.

5. Now, we will use synthesis to evaluate how the area and power of a layer change as the data precision *T* changes. Use your generator to produce four designs (Part 1) with *T* = 8, 12, 16, and 20, while you keep the other parameters constant: (*M*, *N*, *P*) = (8, 8, 1). Then produce two graphs that illustrate: (1) power versus *T* and (2) area versus *T* for these designs. Describe where the critical path is located in each design.

6. Next, we will evaluate how throughput, area, and power scale as *M* changes. Use your generator to produce four designs with *M* = 4, 6, 8, and 10, while you keep *N*=8, *P*=1 and *T*=16. Synthesize each design, and graph: (1) power versus *M,* (2) area

versus *M*, and (3) throughput versus *M*. Does the location of the critical path change as *M* changes?

We will define throughput as the number of data inputs processed per second. To calculate this, you will first determine the maximum number of input data words your system can process per cycle and multiply this by the clock frequency *f*.

We will calculate the *words per cycle* assuming that `s_valid` and `m_ready` are always true. Under these assumptions, for every layer computed, your system processes *N* input words. How many cycles elapse between when you start inputting one group of *N* words, and when you can start inputting the second set of *N* words? (Don't forget: this operation will depend on *M, N,* and *P*). Let *c* represent the number of cycles needed to process these *N* input words. Then, your throughput will be (N/c) words per cycle times *f* cycles per second. (Alternatively, you can measure *c* at the output: how long does it take between when one output vector begins to exit the system, and when the next one exits? This should give the same answer at the input or the output.) In your report, include the values of *c* that you found for each design, and explain how you found them.

7. Now, we will repeat the previous question while *N* changes. Set *M*=8, *P*=1 and *T*=16. Synthesize each design, and graph: (1) power versus *N,* (2) area versus *N*, and (3) throughput versus *N*. Does the location of the critical path change as *N* changes?

8. Evaluate how the designs change when you increase parallelism, by evaluating *P*=1, 2, 4, 8, 16, with *M*=16, *N*=8, and *T*=16. Graph: (1) power versus *P,* (2) area versus *P*, and (3) throughput versus *P*.

   As parallelism increases, the designs should get faster but also more expensive. Which of these designs is most *efficient*? Justify your answer quantitatively.

9. In Part 2, we defined parallelism for this system as having *P* parallel multiply-accumulate units operating on *P* outputs concurrently. However, this limits us to using only *N* multipliers per layer. If you wanted to parallelize further, what could you do?

10. In Part 3, you were tasked with figuring out how to best optimize the parallelism parameters for your three-layer network, given a multiplier budget *L*. Explain how you did this, and why this is a good solution. Do you see any drawbacks to your approach?

11. Lastly, we will evaluate the optimization method you developed for Part 3. For this experiment, set *N*=4, *M1*=8, *M2*=12, and *M3*=16. Set *T*=16. Then, generate and synthesize five different designs, with *L* = 3, 10, 20, 30, and 36. Graph (1) power versus *L*, (2) area versus *L*, and (3) throughput versus *L*.

    When you calculate throughput here, be careful to measure the time once the system has filled the pipeline. In other words, your design is a large pipeline; at the very start of execution, the pipeline is entirely empty, so the design may be faster at this time than during normal operating conditions. A good test is to also measure the amount of time that elapses between the system producing outputs. If you count

that a new input vector can start every *c* cycles, you should also see that the system should produce a new *output* vector every *c* cycles as well.

12. In Part 3, your system assumed it would produce a network with exactly three layers. How would you adapt your generator to create a system with an arbitrary (user-specified) number of layers? What challenges would it create?

9. 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.

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 8, at 11:59PM. No extensions to this deadline will be possible.