ESE-507 Advanced Digital System Design and Generation
Fall 2021

Project 3: Hardware Generation Tool

Start: 11/1/21; Milestone due: 11/17/21 11:59PM
Final project due: 12/4/21 11:59PM

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 added memories and a control module to construct a system that convolved two vectors.

The goal of Project 3 is to extend these ideas and create a piece of software that flexibly generates hardware to accelerate the evaluation of multiple layers of convolutions with a non-linear function called a ReLU ("rectified linear unit") after each convolution.

In Part 1, your software will take as input parameters that describe the vector dimensions and the input data precision. Your software will then generate the corresponding design in SystemVerilog. Part 2 adds parallelism, where your system will compute multiple outputs concurrently.

Then Part 3 will extend this idea to generate a piece of hardware for a multi-layer convolution system, where three modules generated using your solution to Part 2 are composed together to form a larger system. We will refer to each of these three modules as a "layer." 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 to ***maximize system throughput***.

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.

You will be provided with a testbench generator and test scripts that will check your work. These are 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 7)

- 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 7

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.

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 will continue to work with your partner from Projects 1 and 2 (or continue to work alone). **You may not share code with others (except your partner). This means you may not allow others to see your code, nor may you read others' code (for this or related projects). All code will be run through an automatic code comparison tool. Plagiarism will result in a score of zero on the assignment for all involved parties.** If you have questions as to what is acceptable, please come to office hours or send Prof. Milder email to ask for clarification.

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

Point Breakdown
1. Milestone [10 points]
2. Parts 1 and 2 [40 points]
3. Part 3 [30 points]
4. Report, analysis, and code quality [20 points]

Getting Started
Section 2 discusses ways that you will be extending your work from Project 2. Sections 3–5 describe Parts 1, 2, and 3 of the project. Briefly, the goal of part 1 is to adapt your design from Project 2 (with some changes) into a generator. Part 2 adds parallelism to the design. Lastly, in Part 3 you will extend your generator to connect three layers of convolutions together, aiming to optimize the speed of the system given a limited number of multipliers. Section 6 describes the milestone submission (Nov. 17), and Section 7 describes the final evaluation and report. Section 8 gives details of the submission process.

2. Initial Changes from Project 2

In Project 2 Parts 1–3, you built systems that convolved vectors *x* and *f*. You will first be making a some relatively small modifications to this system:

1. **Constant values for *f*:** In Project 2, you took the *f* vector as an input to the system. In Project 3, you 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. **Bitwidth:** In Project 2, we assumed all inputs were 10-bit values, and we increased the precision based on the computations that were performed. Here, for simplicity your system will use the same number of bits for all inputs and outputs. This value will be a parameter. Obviously, this can lead to the situation where your computed value grows larger than the allotted number of bits. You will use *saturation* to address this problem.

3. **Saturating Arithmetic:** Rather than allowing your multiplier or adder to overflow, you will use *saturating arithmetic*, similar to Project 1 (although not exactly identical). Saturating arithmetic simply means that any time the accumulated value grows too large for your system to represent, it must detect this and "saturate" by replacing the value with the maximum representable value. (Similarly, a value can become *too negative* to represent. In this case, you would saturate to the most negative number you can represent.)

   Example: Imagine your system operates on four bits. Therefore, the range of representable values will be:

   1000 → –8
   0111 → 7

   Now, anytime your multiplier or adder should produce a value greater than 7, it should replace its result with 7; anytime it should produce a value less than –8, it should instead replace it with –8.

   In Project 1, it was only possible for your system to saturate after the accumulator. Here, it will also be possible to saturate after the multiplier.

4. **Rectified Linear Unit (ReLU)**: The ReLU function is a simple nonlinear function that will be applied on each individual value of y computed by your system. ReLU simply replaces negative values with 0:

   ReLU(y) = max(0, y)

   If y is positive, then ReLU(y) = y. If y is negative, then ReLU(y) = 0.

   ReLU is commonly used as the activation function in neural networks. For more information about the connections between this project and neural networks, please see class from Nov. 1 (Topic 14).

   The systems produced by your generator will perform a ReLU function on each output of each convolution. 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, your logic simply replaces it with 0.

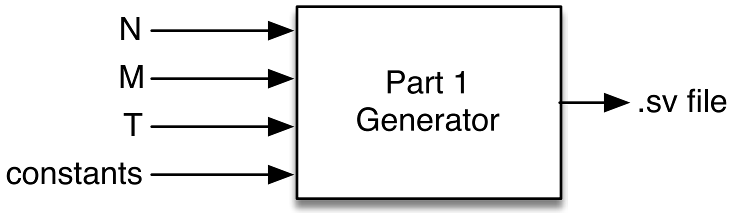
3. Part 1: Basic Generator for a 1D Convolution Layer

The goal of Part 1 is to build a hardware generator that produces a SystemVerilog design for a convolution (as defined in Project 2), controllable by a number of parameters. These parameters are:

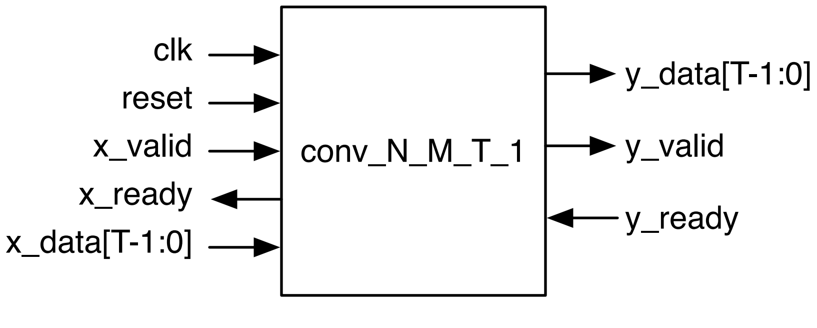
1. **Vector sizes *N* and *M*.** As in Project 2, our input vector *x* will have length *N*, and our filter will have length *M*. This means that our output vector *y* will have length $L = N\text{-}M\text{+}1$. Necessarily, N ≥ M, but to simplify things slightly, you can assume that $N \geq M + 3$.

2. **Bit-width *T*.** The parameter *T* represents the number of bits used as a datatype everywhere in your system. Assume that $8 \leq T \leq 32$. (Note that unlike in Project 2, we will use the same number of bits everywhere, for simplicity.) As described above, our accumulator will use saturating logic to make sure that the result does not grow larger than *T* bits.

3. **Values for the *f* vector.** Your generator will read in a text file that contains the constant values your system should use for *f*. This text file contains one integer (in base 10) per line, and a total of M lines (giving the M values of the vector). Instead of having to read these values as input, your design will include a ROM that is pre-built to hold these values. 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 the system does not need to take *f* as input, so it only has a single stream input port (for the *x* vector).



For part 1, your generator should produce a design whose top-level module is named `conv_N_M_T_1`, where *N, M*, and *T* are replaced with their actual parameter values, e.g., `conv_16_4_16_1`. Use the following port declarations, e.g., if *N*=16, *M*=4, and *T*=16:

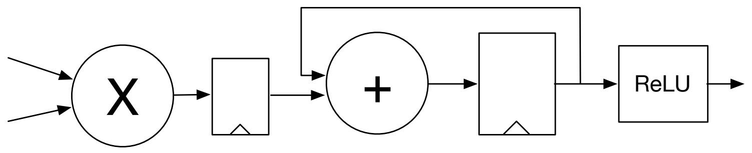
```
module conv_16_4_16_1(clk, reset, x_data, x_valid, x_ready,
                      y_data, y_valid, y_ready);
    parameter T=16;
    input clk, reset, x_valid, y_ready;
    input signed [T-1:0] x_data;
    output signed [T-1:0] y_data;
    output y_valid, x_ready;
```

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

```
    output logic signed [T-1:0] y_data;
    output logic y_valid, x_ready;
```

as needed.

In Project 1, you built several varieties of pipelined multiply-accumulate units. (Many of you did this as well in Project 2.) Here, your system should be based on a simple pipelined datapath with one extra register between the multiplier and adder:



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 vector *f* as input because it is a constant stored in ROM. So, your system's input data will simply be an *N*-element input vector *x*, and its only output data will be the *L*-element output vector *y*.

Assume that the reset signal is positive-asserted and synchronous.

Memories

Your system will hold the input vector *x* in the same memory provided from Project 2. (That is: synchronous reads and writes with a single address port.) **You may not modify the memory in any way.**

Your system will hold the filter vector *f* using the ROM structure seen in the example code.

Saturation

In Project 2, your design used saturation after the adder, but not after the multiplier. That made sense in Project 2, because the multiplier's inputs were 10 bits and its outputs were 20 bits—it could not overflow. Now, your multiplier's inputs and output will all be *T* bits, so it is possible to overflow on the multiplier's output as well as on the adder's output. Your system should account for this by also saturating the multiplier's output values before they go to the adder.

Generator Behavior

Your generator must be executable entirely from the command line on one of our lab machines. If you create the generator using any other computer or programming environment, it is your responsibility to provide a way to run your generator from a lab computer (`cadsrvX` or `labXX.ece.stonybrook.edu`). The generator must take several parameters at the command line. Note that the provided C++ reference code is already set up to handle these parameters for you.

The first parameter specifies the mode. Use mode=1 for Part 1, mode=2 for Part 2, and mode=3 for Part 3. In mode 1, your generator will take as input the following parameters in the following order:

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

Where `N`, `M`, and `T` are the parameters described above, and `const_file` is the name of the file that stores your layer's *f* constants. (Note that the first parameter value 1 indicates we are running in mode 1.)

The generator will store the result in a file whose name matches the top-level module: `conv_N_M_T_1.sv`, where the letters N, M, and T are replaced with the actual parameter values.

For example, if you want to implement a layer with an input vector of length 16 and a filter of length 4, with 20 bits, and with *f* constants stored in a file called `const.txt`, you would run:

```
./gen 1 16 4 20 const.txt
```

The result would be stored in `conv_16_4_20_1.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 large variety 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 the project, 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 *f*.

To run the testbench generator, run

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

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

This will produce four files:

• `tb_conv_N_M_T_1.sv`	the testbench file
• `tb_conv_N_M_T_1.in`	the inputs to test
• `tb_conv_N_M_T_1.exp`	the expected results
• `const_N_M_T_1.txt`	the constants to give your generator

Then, you would generate the accompanying code with:

```
    ./gen 1 N M T const_N_M_T_1.txt
```

(where again M, N, and T are replaced with your parameter values). This will produce your `conv_N_N_T_1.sv` file.

Then, to simulate:

```
    vlog conv_N_M_T_1.sv tb_conv_M_N_T_1.sv
    vsim tb_conv_N_M_T_1
```

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

The simulation will report any errors.

Important: I strongly encourage you to develop the generator using the lab computers. The testbench generator relies on having a standard Linux command line. It has been tested to work on Linux, and it should probably work on MacOS. However, you likely will not be able to run it under a standard Windows development environment without some extra setup. If you prefer to work on a local Windows computer instead of the lab Linux machines, you may need to first generate the testbench on the lab computer, and then copy its files to your Windows computer.

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:

```
    ./testmode1 N M T
```

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

For example, if you want to implement and test a convolution with an input x of length 16 and a filter of length 4, using 20 bits, you would run:

```
    ./testmode1 16 4 20
```

These test scripts rely on the Linux shell, so you should use the lab Linux computer if you want to use them.

Getting Started

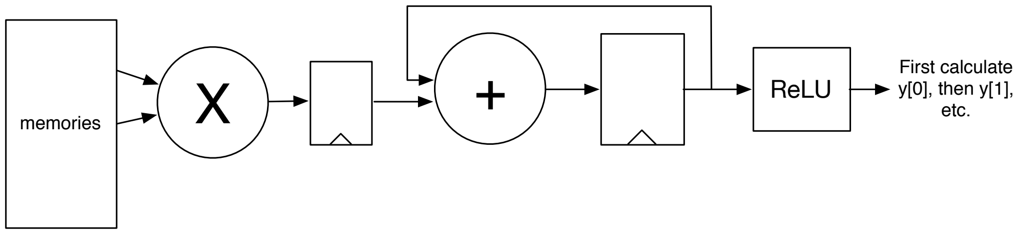
Start by copying our handout code to your work directory. (Please make sure you set the permissions correctly so your work is not visible to others.) Then, compile the code and try running it. Look at the output file that is produced. Look through `main.cc` and understand how the code is structured. 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.

4. 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 3 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:

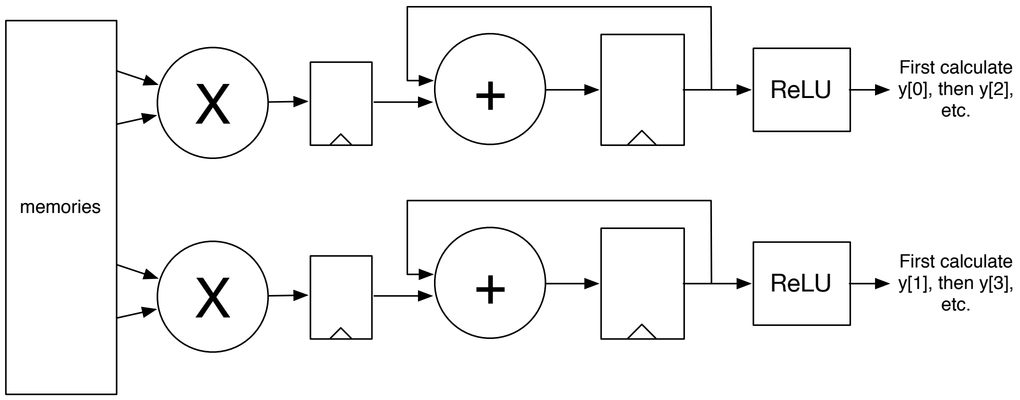
P=1



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:

P=2


memories
X
+
ReLU
First calculate y[0], then y[2], etc.
X
+
ReLU
First calculate y[1], then y[3], etc.

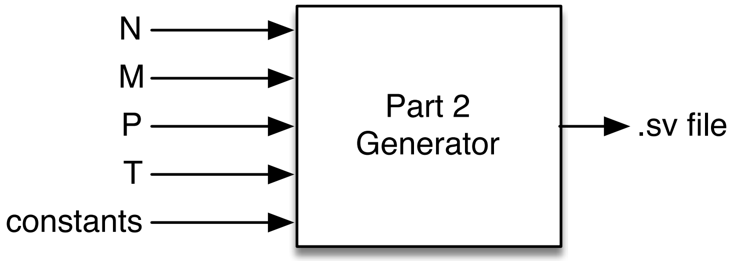
Now the first accumulator calculates y[0] while the second accumulator calculates y[1]. Then, they will compute y[2] and y[3] concurrently, and so on.

Like Part 1, you must use the provided memory structure (synchronous reads and writes with a single address port used for both read and write). 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 *L*/*P* is an integer. (Recall: *L = N-M+1* represents the number of output values in the vector your layer produces.) Therefore, if your system outputs *L*=16 words, then legal values of *P* are 1, 2, 4, 8, and 16. However if *L*=13, the only legal values of *P* are 1 and 13.

Generator and Testbench

For Part 2, you will now use mode=2 when running the generator. In this mode, you will also provide the *P* parameter.



You will run the generator for Part 2 using:

```
      ./gen 2 N M T P 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 `conv_N_M_T_P`, where the *M*, *N*, *T*, and *P* letters are replaced with their actual parameter values, e.g., `conv_16_5_20_3`. Use the following ports declarations, e.g., if *N*=16, *M*=5, *T*=20, and *P*=3:

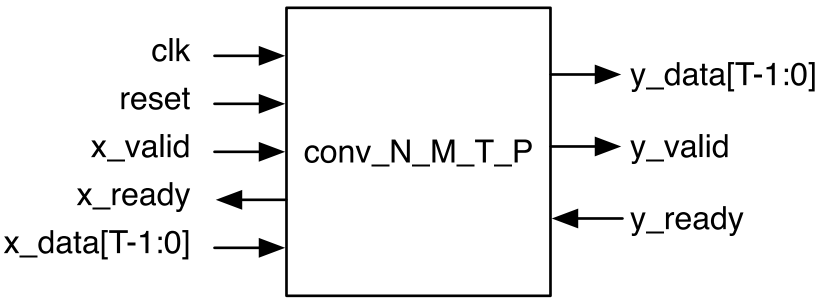
```
module conv_16_5_20_3(clk, reset, x_data, x_valid, x_ready,
                      y_data, y_valid, y_ready);
    parameter T=20;
    input clk, reset, x_valid, y_ready;
    input signed [T-1:0] x_data;
    output signed [T-1:0] y_data;
    output y_valid, x_ready;
```

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

```
    output logic signed [T-1:0] y_data;
    output logic y_valid, x_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 2 N M T P
```

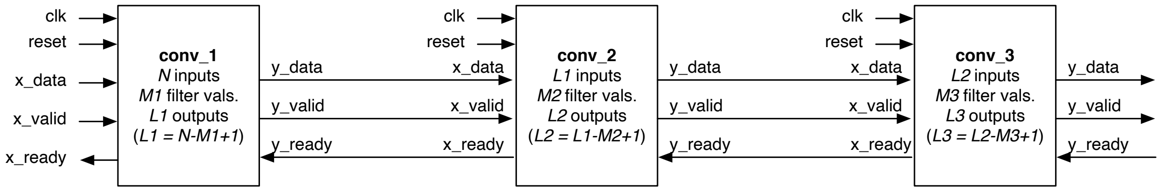
You can run the test script (which again will generate a testbench, generate a design, and simulate the design):

```
./testmode2 N M T P
```

where in both cases M, N, T, and P are replaced with the appropriate parameters.

5. Part 3: Generating and Optimizing a Three-Layer System

Inspired by deep convolutional neural networks[1], we will now construct a system with three layers of convolutions (each with a ReLU after the convolution).



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's input has size *N* and its filter is of length *M1*. Therefore, Layer 1's output will be of length *L1 = N-M1+1*.
- Then, Layer 2 will have input size *L1* and a filter size of *M2*, producing *L2 = L1-M2+1* output values.
- Lastly, Layer 3 will have an input vector of size *L2*, a filter of length *M3*, and it will produce *L3 = L2-M3+1* output values.

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*. (But recall that the

[1] Please see discussion from class topic 14

parallelism must evenly divide the output size for each layer, so *L1/P1*, *L2/P2*, and *L3/P3* must all be integers.)

Assume that all three layers have the same values of *T*.

Measuring Performance as Throughput

The system we have described is a big pipeline—values will flow out of one layer into the next. Once Layer 1 is done computing a convolution, its resulting output vector will flow into Layer 2, and so on. Ideally, all three layers will be able to perform useful computations concurrently. While Layer 1 is processing a new input vector, Layer 2 can concurrently process the *previous* result that Layer 1 computed. At the same time, Layer 3 can concurrently process the previous result that Layer 2 computed.

So, we will evaluate the performance of this system based on its best-case *throughput*. Specifically, we will measure throughput as the number of input words the system can process per second (with best-case assumptions about `x_valid` and `y_ready` signals). See Questions 4 and 9 in Section 7 below for some more information about measuring throughput.

Optimization Problem

Notice that the throughput of each layer will now depend on its dimensions and its parallelism. This leads to an opportunity for optimization: given a *multiplier budget B*, how do you choose the *P1*, *P2*, and *P3* parameters in order to maximize overall throughput while using no more than *B* multiply-accumulate units? Necessarily, $B \geq 3$ (since there's no way for us to produce a layer with less than one multiply-accumulate unit). Similarly, $B \leq (L1+L2+L3)$ because this would enable the maximum parallelism for each layer.

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

Your optimizer does not need to allocate all *B* multipliers. In fact, it should only use multipliers that will improve the throughput of the design. (In other words, only use the whole budget if each multiplier you add improves throughput.)

Think carefully about your optimization approach and how you will evaluate it. Specifically: how will you determine if your optimizer made good choices for P1, P2, and P3?

Generator

For this task, your generator will operate in mode=3, where you will be specifying the parameters for a three-layer system:

```
      ./gen 3 N M1 M2 M3 T B 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 `net_N_M1_M2_M3_T_B`, and the result should be stored in a file called `net_N_M1_M2_M3_T_B.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 *f* matrix. Now, this file will need to store ***all*** the constants for all three convolutions. The first layer will require the first *M1* elements, the second layer will use the next *M2* elements, and the last layer will use the final *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:

```
./testgen 3 N M1 M2 M3 T B
```

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

```
./testmode3 N M1 M2 M3 T B
```

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

6. Milestone: Due 11/17/21, 11:59pm

The milestone is an intermediate set of tasks, which will be due on 11/17. For the milestone, you are required to finish Part 1. This is a halfway through the project's allocated time. ***Ideally, by this date you should also have made progress on 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):
  - *N*=16, *M*=6, *T*=16
  - *N*=19, *M*=11, *T*=20
  - *N*=30, *M*=9, *T*=11
  - *N*=39, *M*=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").

7. 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. Describe how you designed your control logic. How did you structure the design so that it can be changed as the *M* and *N* parameters change? Explain how the resulting hardware changes as these parameters grow.

2. 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?

The evaluation will ask you for results from a specific set of designs. However, you should test your generator with many different sets of parameters to ensure it works correctly. (This is easy to do; you can simply use the `./testmode*` scripts to test a wide variety of configurations.)

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" or "declaration initial assignment") are *very* big problems. If your synthesis report shows these, make sure to correct the problem and re-synthesize.

3. Now, we will use synthesis to evaluate how the area and power of a single convolution module 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: (*N*, *M*, *P*) = (16, 4, 1). Then produce two graphs that illustrate: (1) power versus *T* and (2) area versus *T* for these designs. (Place *T* on the x-axis of both graphs.) Describe where the critical path is located in each design.

4. 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*=32, *P*=1 and *T*=16. Synthesize each design, and graph: (1) power versus *M*, (2) area versus *M*, and (3) throughput versus *M*. (Place *M* on the x-axis.) 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_{clk}.

We will calculate the *input words per cycle* assuming that `x_valid` and `y_ready` are always 1. 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? Call the result *c* cycles. (Don't forget: this operation will depend on *M, N,* and *P*). Then, your throughput will be (N/c) words per cycle times the clock frequency (f_{clk} 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.**

5. Now, we will repeat the previous question while *N* changes. Use your generator to produce four designs with N=16, 32, 64, and 128. 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?

6. Evaluate how the designs change when you increase parallelism, by evaluating *P*=1, 2, 4, 8, 16, with *N*=96, *N*=65, 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.

7. 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 *L=N–M+1* multipliers per layer. If you wanted to parallelize further, what could you do?

8. In Part 3, you were tasked with figuring out how to best optimize the parallelism parameters for your three-layer design, given a multiplier budget *B*. Explain how you did this, and why your approach is a good solution. Do you see any drawbacks to your approach? ***How did you evaluate whether or not your system worked well?***

9. Lastly, we will evaluate the optimization method you developed for Part 3. For this experiment, set *N*=64, *M1*=33, *M2*=9, and *M3*=10. Set *T*=16. Then, generate and synthesize five different designs, with B = 3, 8, 14, 30, and 50. Graph (1) power versus *B*, (2) area versus *B*, and (3) throughput versus *B*.

   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.

10. In Part 3, your system assumed it would produce a system 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?

11. If you worked with a partner: please explain each partner's contribution to the project. Be specific about what each partner did. (If you worked alone, skip this.)

8. 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. If you built your software on another computer or environment, please make sure you include instructions for how to build and run it on the lab computers. Additionally, please update the `testmode1`, `testmode2`, and `testmode3` scripts to work correctly with your code.

The `hdl/` directory should hold the SystemVerilog files you generated to complete the evaluation (specified in Section 7 of the assignment). Also include your 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 Saturday, December 4, at 11:59PM. No extensions to this deadline will be possible.