Stony Brook University

Project 2: 1D Convolution Engine

Issued: 10/7/19; Due 11/4/19, 4:00PM

Introduction

In this project, you will utilize memories and arithmetic units to build a system that performs a one-dimensional convolution.

In part 1 [40 points], you will construct a system that takes as input two vectors and convolves them, outputting the result. In part 2 [15 points], you will extend this system to work with a larger vector sizes. Then, in part 3 [35 points], you will modify your system to make it faster. In addition to the 90 points described above, 10 points will be awarded based on the quality of your code, comments, and report.

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

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

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

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

Your project will be evaluated on the correctness and efficiency of your designs 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). 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.

Partner

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

- Send an email to peter.milder@stonybrook.edu with the subject "ESE 507 Project 2 Partner Signup"
- Send the email from your @stonybrook.edu email address

- In the body of the email, write both your name and your partner's name. (You don't need to include any other text in the email.)
- CC your partner on the email (using your partner's @stonybrook.edu email address)

After this, you are committed with working with this partner for project 2. (If you want to change for project 3, you may.)

Background

Convolution

Convolution is a mathematical operation that takes as input two signals and combines their values to produce a third signal. Assume you are given two discrete signals x[n] and h[n]. Let's call x[n] our "input" and h[n] our "filter." We can define their convolution y[n] as:

$$y[n] = x[n] * h[n] = \sum_{k=-\infty}^{\infty} x[k]h[n-k]$$

You probably have seen an expression like this in a signal processing class. Here, we will make some reasonable simplifications:

1. We will assume that the input and filter signals are finite (that is, they have a finite number of non-zero values).
2. We will only calculate the values of *y* that correspond to locations where *x* and *h* fully overlap.
3. We will assume that we already have a version of h[n] stored in reversed order, which we will call f[n] (so therefore f[n] = h[-n]).

Let's assume that x[n] is of length N and f[n] is of length M and that N>M. Let L = N–M+1. So, we can now write their convolution as:

$$y[n] = \sum_{k=0}^{M-1} x[n+k]f[k], \quad n = 0, \ldots, L-1 \qquad (1)$$

Or, in pseudocode:

```
for n = 0 to L-1:
    y[n] = 0
    for k = 0 to M-1:
        y[n] += x[n+k]*f[k]
```

For example, let x = [10, –20, 30, 40, –50, 60] and let f = [5, 4, 3]. Therefore N=6, M=3, and L=4. (Note that values of x and f can be positive or negative as seen here.) We would calculate our four output values:

```
y[0] = x[0]*f[0] + x[1]*f[1] + x[2]*f[2] = 10*5—20*4+30*3  = 60
y[1] = x[1]*f[0] + x[2]*f[1] + x[3]*f[2] = —20*5+30*4+40*3 = 140
```

```
y[2] = x[2]*f[0] + x[3]*f[1] + x[4]*f[2] = 30*5+40*4–50*3  = 160
y[3] = x[3]*f[0] + x[4]*f[1] + x[5]*f[2] = 40*5—50*4+60*3  = 180
```

Your system will operate in the following way. First, it will take as input two streams of values corresponding to `x` and to the filter `f`. These will be stored into memory. Then your system will begin executing: it will first compute output `y[0]`, then `y[1]`, and so on. Since each memory will allow you to read a single word per cycle, it will take several cycles to compute each output value.

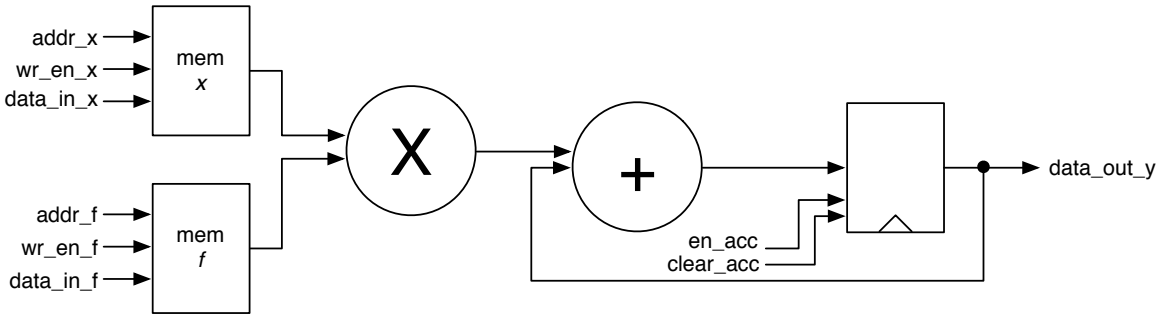

Figure 1. Simplified Datapath Block Diagram.

Consider the simplified datapath block diagram shown in Figure 1. (This is a suggested organization for your system, but you can feel free to make changes.) First, values of x and f will come in on the `data_in_x` and `data_in_f` ports. Your control logic will be responsible for setting the address and write enable signals appropriately so that they are stored in memory correctly. Then, your system will calculate each output following the pseudocode on the previous page. First, it will clear the accumulator register (initializing it to zero). Then, over the next *M* cycles, it will read one value from `x` and one value from `f`, multiply them together, and add them to the running total. After you have multiplied and added `M` times, the system can output a final value of `y` on `data_out_y`, and then repeat the process for all L=N–M+1 values.

This simplified description omits several key concerns. Further details, including the exact sizes of vectors and numbers of bits, as well as the input/output signals and protocol will be discussed below.

Memory

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

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

    parameter                 WIDTH=16, SIZE=64, LOGSIZE=6;
    input [WIDTH-1:0]         data_in;
    output logic [WIDTH-1:0] data_out;
    input [LOGSIZE-1:0]       addr;
    input                     clk, wr_en;

    logic [SIZE-1:0][WIDTH-1:0] mem;
```

```
    always_ff @(posedge clk) begin
      data_out <= mem[addr];
      if (wr_en)
          mem[addr] <= data_in;
    end
endmodule
```

There are several important things to understand about this memory:

- The memory has one read port, one write port, and one address input. All reads and writes are synchronous (that is, they will occur on a positive clock edge).

- Unlike some examples we looked at in class, this module only contains a single address input, which will be used for both reading and writing. (So, you cannot read and write to different locations of this memory at the same time.)

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

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

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

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

Figure 2 demonstrates the timing of reading from the memory. On each positive clock edge, the system samples the value on `addr`. A short time after the edge, it will output the value in memory at location `addr`. In this diagram, `mem[7]` just represents the value stored in address 7 of the memory.

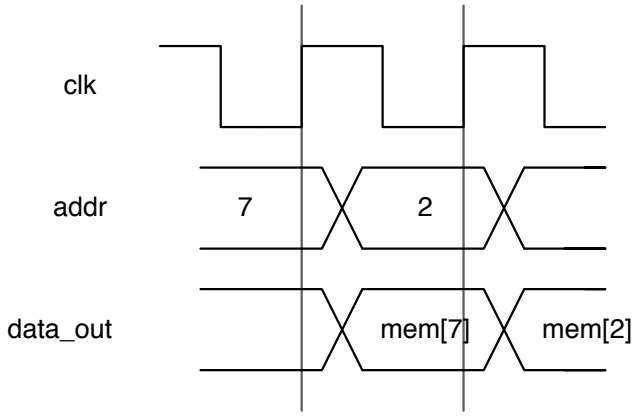

Figure 2. Timing of memory read.

Figure 3 demonstrates the timing of writing to memory. In this example, you are first writing the value 3 to address 6. Then, on the following cycle, no write is performed because

the `wr_en` signal is 0 on the clock edge. Then, value 4 is written to address 1 in the third cycle.

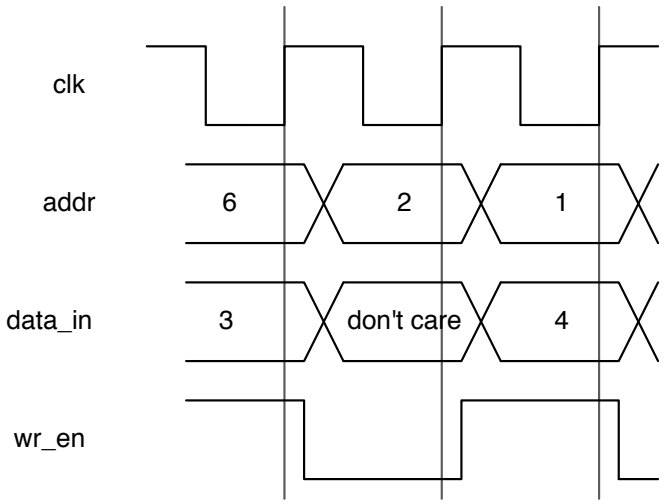

Figure 3. Timing of memory write.

Recall from class that this RTL memory module will not synthesize to SRAM—instead it will produce a memory structure out of flip-flops. If these memories were large, this would be very inefficient. However, the memories you will need in this project will be fairly small, so we will simply let the logic synthesis tool implement them using registers.

Input/Output Protocol

For this and future projects, we will use a simplified variation on ARM's AXI4-Stream Protocol. Shown in Figure 4, this is a simple synchronous protocol that allows a sender (called a "master") to transfer data to a receiver (called a "slave") when both sides agree.

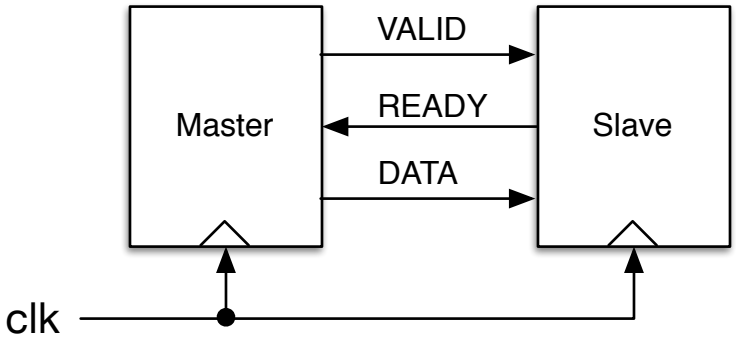

Figure 4. Handshaking signals

The master asserts the VALID signal when it has placed valid data on the DATA signal. The slave asserts the READY signal when it is capable of consuming that data. On any positive clock edge, data is transferred if (and only if) both the VALID and the READY signals are asserted. (No data will ever be transferred unless both are asserted.) Note that both master and slave share a common clock.

The diagram (Figure 5) and accompanying table (Table 1) on the following page illustrate this process.

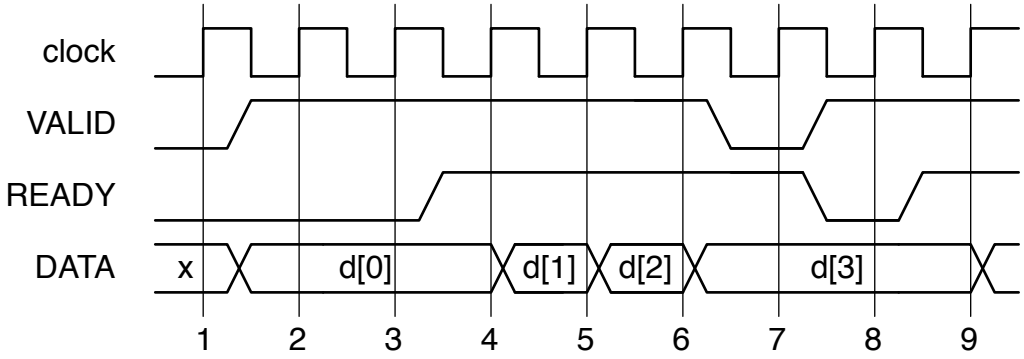

Figure 5. Handshake Timing Example

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

Table 1. Handshake Timing Example

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

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

Basic System Architecture

As you can see from expression (1) and the discussion above, each output can be computed using a multiply-and-accumulate (MAC) unit that consists of a multiplier, an adder, and a register. The MAC will be fed with appropriate values from the memories to perform the required operations. For example, you would calculate `y[0]` by first clearing the accumulation memory, and then feeding the unit inputs `x[0]` and `f[0]` on the first cycle, `x[1]` and `f[1]` on the second cycle, and so on.

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

Figure 6 shows the top-level view of your design. In addition to inputs `clk` and `reset`, the system has three stream ports: a slave port for `x`, a slave port for `f` and a master port for `y`. Recall that each stream port consists of three signals: data, valid, and ready (see "Input/Output Protocol" above). The system's data inputs are 8-bit signed values, and the data output is `t`-bits, also signed. (The value of `t` will be discussed below.)

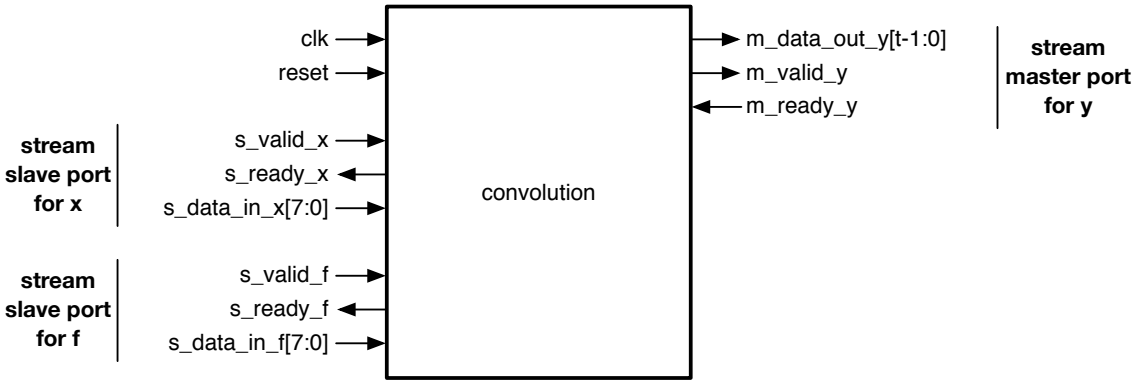

Figure 6. Top-level design. See above for information about valid/ready signals.

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

After your system finishes a computation, it should be able to immediately begin taking a new set of inputs for the next computation. Remember, a slave port should only assert its `s_ready` signal when it is capable of taking new input in. Similarly, when your system is outputting results, its master port needs to wait until `m_ready` is asserted for the result to be successfully received.

Part 1: Small Convolution [40 points]

In Part 1, you will get started by building a system for N = 8, and M = 4, that is, the `x` vector will have length 8 and the `f` vector will have length 4. This means the output vector will have length 8–4+1 = 5.

Your system will use the input/output protocol described above, with x vector inputs received on one slave port, and f vector inputs received on the other. Once all values from both vectors are received, your system can compute and output the 5 values of y.

Since the input values are 8 bits each, we will size our multiplier's output to be 16 bits and we will make the adder's accumulation path 18 bits. (Think carefully: *why is 18 bits the right value for this system?)*

Here you will build a datapath with two memories, a multiplier, and an accumulator (adder plus register). Figure 7 shows a rough idea of what this system should look like. You can feel free to design your own datapath, but for Parts 1 and 2 it is required you follow this basic structure: two memories, one multiplier, and one adder. Each block labeled `mem` is an instantiation of the module named `memory` from above. (Note that you will need to determine the correct parameters for these modules, as well as the bits needed for the address lines.)

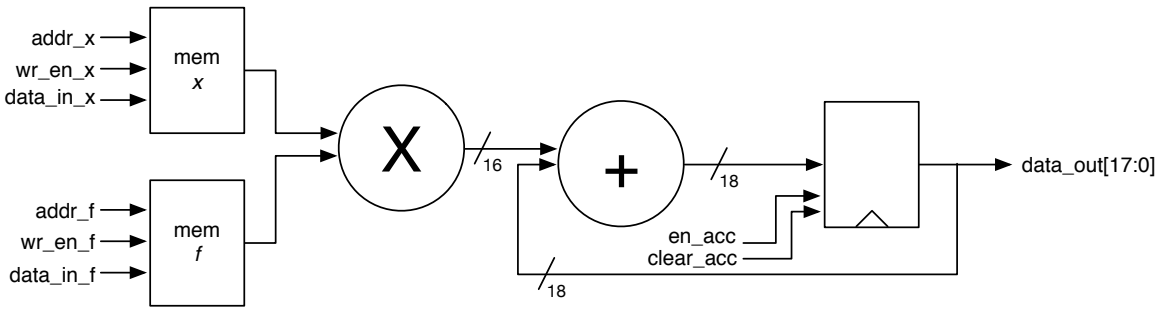

Figure 7. Datapath structure.

Note that many of the datapath's inputs are control signals. These signals (`addr_x, wr_en_x, addr_f, wr_en_f, clear_acc, en_acc`) will need to be generated by a `control` module. Your control module will the contain control logic you require (e.g. counters, FSMs, etc.). Figure 8 shows the interconnections between your datapath and the control module. Feel free to add, remove, or change the control signals that your system uses. The control system is the most complex portion of the design. Think carefully about how you want the control module to behave and interact with the datapath.

I recommend you begin designing your control system by breaking it down into three cooperating control blocks: one that handles loading data into memory x, one that handles loading data into memory f, and one that controls reading data from those memories and controlling the accumulator and output signals. These three control blocks can then cooperate (e.g., when both data-loading controllers signal that input vectors x and f have been fully loaded, the "compute" controller can begin working).

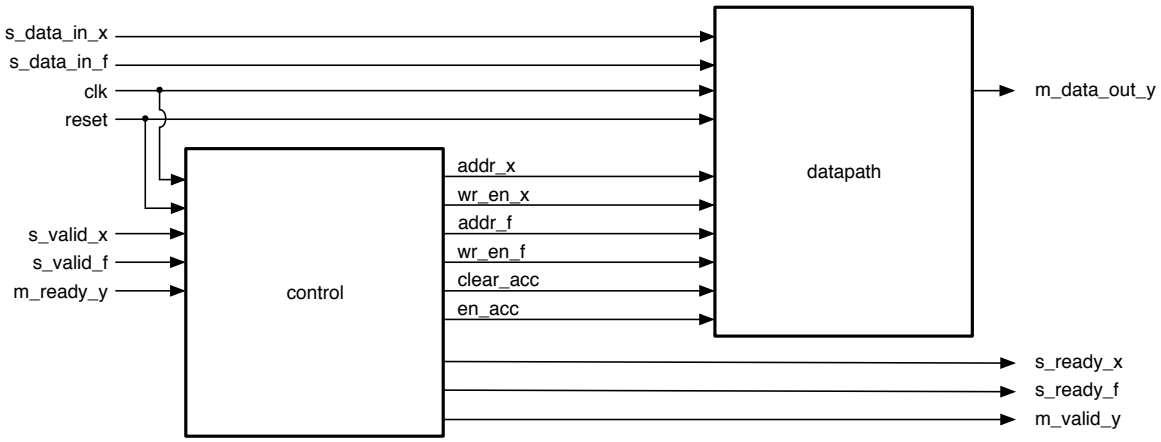

Figure 8. Interconnections between control unit and datapath.

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

```
module conv_8_4(clk, reset, s_data_in_x, s_valid_x, s_ready_x,
                s_data_in_f, s_valid_f, s_ready_f, m_data_out_y,
                m_valid_y, m_ready_y);

    input clk, reset, s_valid_x, s_valid_f, m_ready_y;
    input signed [7:0]   s_data_in_x, s_data_in_f;
    output               s_ready_x, s_ready_f, m_valid_y;
    output signed [17:0] m_data_out_y;
```

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

```
    output logic               s_ready_x, s_ready_f, m_valid_y;
    output logic signed [17:0] m_data_out_y;
```

as needed.

Example Testbench. To help you get started, we will provide you with two testbenches: one is a very small simple testbench that you can use to check that your system's timing is correct. The other is a large random testbench that you can use to test more thoroughly. In both cases, to help simulate the fact that the `s_valid_*` and `m_ready_y` signals can change at any time, these testbenches use a random number generator to set these values. On every cycle, the testbench will randomly decide whether or not `s_valid_x`, `s_valid_f`, and `m_ready` are set. This randomization makes the test much more robust.

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

```
vsim -sv_seed 42 [the rest of your vsim commands go here]
```

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

This testbenches are available at:

```
/home/home4/pmilder/ese507/proj2/part1/part1_simple_tb.sv
/home/home4/pmilder/ese507/proj2/part1/part1_random_tb.sv
```

To use the random testbench, you will also need to copy:

```
/home/home4/pmilder/ese507/proj2/part1/random_in.hex
/home/home4/pmilder/ese507/proj2/part1/expected_out.hex
```

Note: when you compile these testbenches with "vlog" you will get a warning message:

```
** Warning: part1_random_tb.sv(46): (vlog-2254) SystemVerilog
testbench feature
(randomization or coverage) detected in the design.
These features are only supported in Questasim.
```

This warning is safe to ignore.

Your tasks for Part 1 are:

1. Create this design in SystemVerilog. For ease of debugging and future extension, I suggest you structure your design with a top-level module (`conv_8_4` as above), a control module, and a datapath module. Use `conv_8_4.sv` as the name of your top-level file.

2. Test your design. The small testbench we will provide for you is a simple place to get started, and the random testbench we provide is good for thorough testing. However, you likely will want to have some medium tests between these two. In your report, make sure you explain how you tested and what you learned.

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

4. In your report, include the following:

    a. How many arithmetic operations are required to convolve a vector x of length N=8 with a vector f of length M=4 (using our slightly simplified version of convolution)? Then, generalize this: how many arithmetic operations are required to convolve a vector of length *N* with a vector of length *M*?

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

    c. In this design, we take 8-bit inputs; the multiplier's product is 16 bits; the accumulator is 18 bits. Is this sufficient to guarantee the system cannot overflow? Explain why or why not.

d. Explain how you verified your system. Did you find and solve any problems? Did you write any other testbenches besides those provided to you? (If you did, please include them with your submission.)

e. Report the area, power, frequency, and critical path location you determined from your synthesis report. Explain the critical path location descriptively; in other words, explain where the path flows through your design and why it makes sense that it is the critical path.

f. From your simulation (and your understanding of your design), determine how many clock cycles the system takes to load one set of inputs, compute one convolution when the vectors are length 8 and 4, and output the result, ***assuming that the testbench does not stall your design*** (that is, the testbench ensures that s_valid_* and m_ready are always asserted). Count from the first cycle of loading the input until the last cycle of outputting the result. Then, multiply this by the fastest clock period you could reach to find the minimum delay of the system (in nanoseconds).

g. A joint metric that combines the effects of area and speed in a single value is the *area-delay product*. The area-delay product is found by multiplying the area of the system times its delay. (Since these are both metrics that we want to minimize, lower area-delay products are better than higher ones.) Calculate the area-delay product of your system.

h. Based on the synthesis power estimate, how much energy is consumed by your system while computing one convolution (using the delay you found in part f)?

i. We can define the *arithmetic operation count* as the number of useful additions and multiplications required to perform one convolution. What is the operation count for your system? How much energy does your system consume *per arithmetic operation*?

Part 2: Larger Convolution (128 by 32) [15 points]

In Part 2, you will adapt your design from Part 1 to make a design with N=128 and M=32. That is, your input vector x is now of length 128 and f is of length 32. Then you will reason about how the design will change as N and M increase.

Use module name `conv_128_32` for the top-level module of this design and use `conv_128_32.sv` as your top-level filename. Create this design, update your testbench, simulate your design, and synthesize it. ***Change the adder's output (accumulator) to 21 bits (instead of 18). This will also mean the system's output data port will be 21 bits.*** In your report explain how the design changed. How difficult was it to change your control module?

We have again provided testbench modules for you. Please see:

```
/home/home4/pmilder/ese507/proj2/part2/part2_simple_tb.sv
```

```
/home/home4/pmilder/ese507/proj2/part2/part2_random_tb.sv
/home/home4/pmilder/ese507/proj2/part2/random_in.hex
/home/home4/pmilder/ese507/proj2/part2/expected_out.hex
```

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

Repeat steps e–i from Part 1 for your new design, including your responses in your report. Also, answer the following question: *Why did we change the accumulator from 18 bits to 21 bits? Is this sufficient to guarantee the system cannot overflow?*

Part 3: Delay-Optimized System [35 points]

Now, you will take your design from Part 2 and modify it to be as fast as possible, while keeping the input/output ports and timing as specified in the previous sections. Name your top-level synthesizable module `conv_128_32_opt` and use `conv_128_32_opt.sv` for your top-level filename. You may change the internals of your design in any way possible, but the input/output behavior must not deviate from this specification. (In other words, your system must still have two slave ports (s_valid/s_ready/s_data), one each for f and x for input and one master port (m_valid/m_ready/m_data) for output y.

Some ideas you may want to pursue:

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

You may use DesignWare components for adders and multipliers, but you may not use the DesignWare multiply-and-accumulate unit. If you choose to use any DesignWare components, you will need to include them in your simulation (see slides from end of Class 12, October 7).

this design, update the testbench, simulate your design, and synthesize it. Note that the exact same testbench from Part 2 will still work, as long as you change the module instance to instantiate your new design. In your report explain:

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

b. Collect the information requested in parts e–i from Part 1 for your new design and include them in your report.

c. Your new design performs the same computation as your design in Part 2, but it should be faster, larger, and consume higher power. Compare its area-delay product and energy per arithmetic operation with your Part 2 design. In these metrics, is your faster/larger design better or worse than your previous design?

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

e. Because you are constrained by the number of input and output ports, the maximum speed of your design here is limited. What could you do to make a design even faster, if you were allowed to change the input/output timing and ports? Estimate (quantitatively) how fast such as system could be.

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

Code and Report Submission

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

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

2. **Report**
   Your report should include the information requested above. Your report should be electronically produced (not handwritten) and neatly formatted. Include your report in the electronic hand-in with your code **(as a PDF file only)**. (An interesting note that some people failed to understand in Project 1: a PDF file is different from a Word .doc or .docx file!)

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