ECE 526L Lab9

Experiment #8, Sum of Products

The diagram below represents a sum of products circuit, which is a method of digital filtering. It is one of many hardware algorithms for this function.

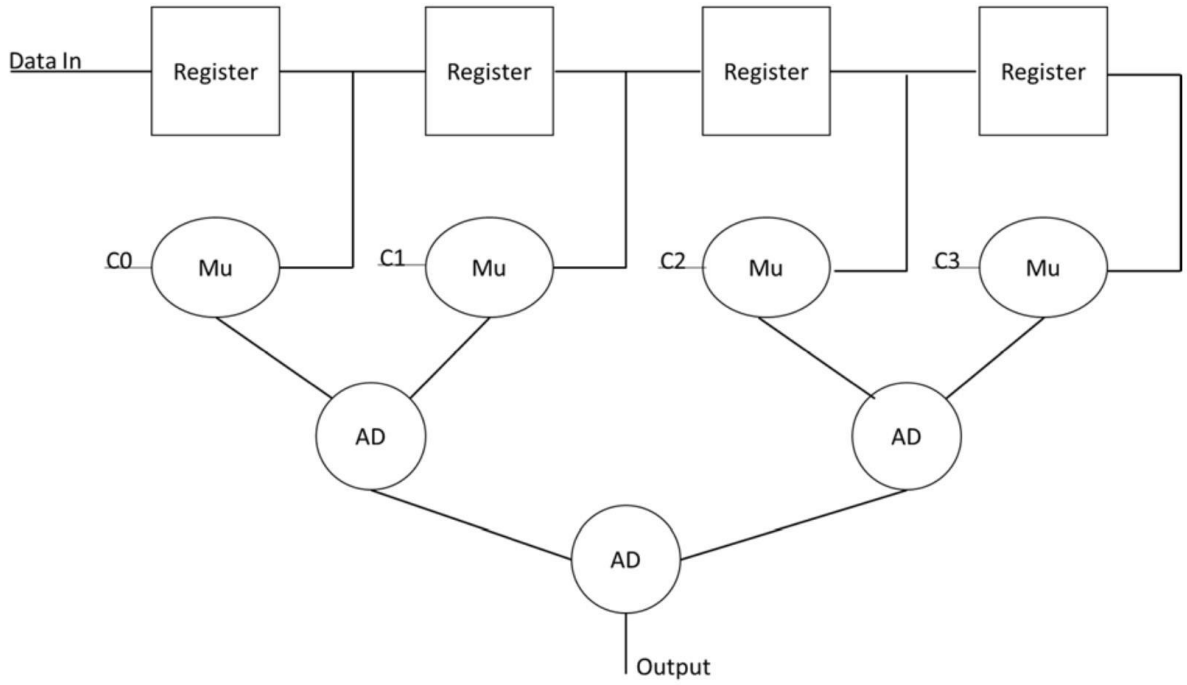
Using three levels of hierarchy, create a Verilog circuit description of this device. Data paths are to be scalable. The default value for both the coefficient (Cx) and Data In widths is to be four bits. Coefficients as well as Data_In are primary system inputs.

At the lowest level, you will thus have three design modules: a register, a multiplier and an adder. Each is scalable.

The second level consists of one instance of an adder, two instances of multipliers and two instances of registers.

The top design level has two instances of the second level and one more adder. The top level is to be instantiated in test benches. While not required, you may find it helpful to verify lower level designs independently before integrating all design levels into the top level design and test bench.

In the diagram below, ovals are multipliers. Circles are adders. Squares are registers. There is a **clock** primary input to the registers which is **not shown** in the diagram.



At each stage of addition, the width needs to increase by one bit. The multiplier outputs need to be twice the data input width and the adders need to increase by one bit at each stage to avoid any potential data loss.
Use behavioral modeling to create a scalable register, multiplier and adder. Instantiate two registers, two multipliers and one adder to create the second level of hierarchy. The third level will consist of two instances of the second level plus one instance of the adder. Adjust widths as needed to prevent data loss by overriding the default parameter values for each instance.

Do not use **defparam** anywhere in your code.

Write a non-exhaustive testbench to test the functionality of your device. Select a small number of robust test vectors. *n.b*. "small" no more than ten. Your test strategy should discuss why you chose the particular vectors you did. Why do you think the chosen set is robust? What, if anything, is not tested by the chosen set?

Write an exhaustive testbench to test your device. Compare your output to the output of a non-hierarchical sum of products. If they agree, continue testing until all inputs have been presented. If no errors were found, indicate the test was successful. If they do not agree, indicate the test has failed and show the operands and the expected and actual outputs. Stop testing when the first error is found or when all test vectors have been successfully applied.

To prove your exhaustive testbench will find an error, use a **force** assignment to override the behavioral code's sum sometime after 3⁄4 of the test vectors have been applied. Use the **`ifdef** and **`endif** compiler directives to include or exclude the code that forces an error.

In your report, include log printouts from both the error-free and forced error cases. Use **$monitoroff** and **$monitoron** commands to show only the first dozen or so test vectors (and results) and the last dozen or so for the error free case. For the forced error case, use them to show the first dozen or so test vectors and about a dozen test vectors before and including the forced error.

For the error-free case, include waveform printouts showing the start and end of the test in sufficient detail to verify the applied signals and how they change. You may, if you wish, submit one or more "overview" printouts as well.

For the forced error case, include waveform printouts of the end of the test in sufficient detail to see when and how the forced error is applied and how the modules react.

To make your output easy to analyze, use decimal numbers and show input values, output values and expected output values. When errors are encountered, your test bench should at a minimum indicated when (in simulation time) the error occurred, the expected value and the received value.

Explain why you included or did not include a reset in your design. The design specification does not address this issue.

Note: this design does not contain any inout (bidirectional) ports. The output of the second register is used internally as an input to its multiplier and as a signal to the third register, but that does not make it bidirectional.