Objective:

In this lab, you will be implementing a simple pipelined circuit. The circuit will utilize one 32-bit block RAM to continually provide 4 8-bit inputs per cycle, and one 17-bit block RAM to store an output each cycle. In software, you will initially transfer data from the microprocessor into the input block RAM, specify the size of the input, start the circuit, and then wait for completion, at which point the software will read data from the output blockRAM and output it to the screen.

You will again be working in groups of 2 on this project. Unless you are an EDGE student or have special permission from the instructor, you must work in a group of 2. Note that those in the overflow section are *not* EDGE students, and must work in a group of 2. Please only submit once per group.

EDGE INSTRUCTIONS:

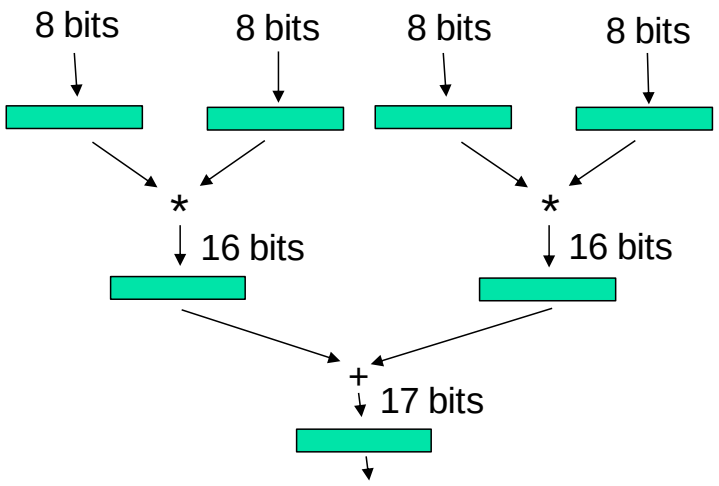
EDGE students must complete the entire lab, but will receive more partial credit than people working in groups.

Part 1 – VHDL

1) Download the provided code. The directory structure is the same as the previous lab.
2) Read the following pseudocode, which is the functionality that your circuit will implement**.** Note that your actual code will look nothing like this. This pseudocode is simply intended to help you understand the functionality.

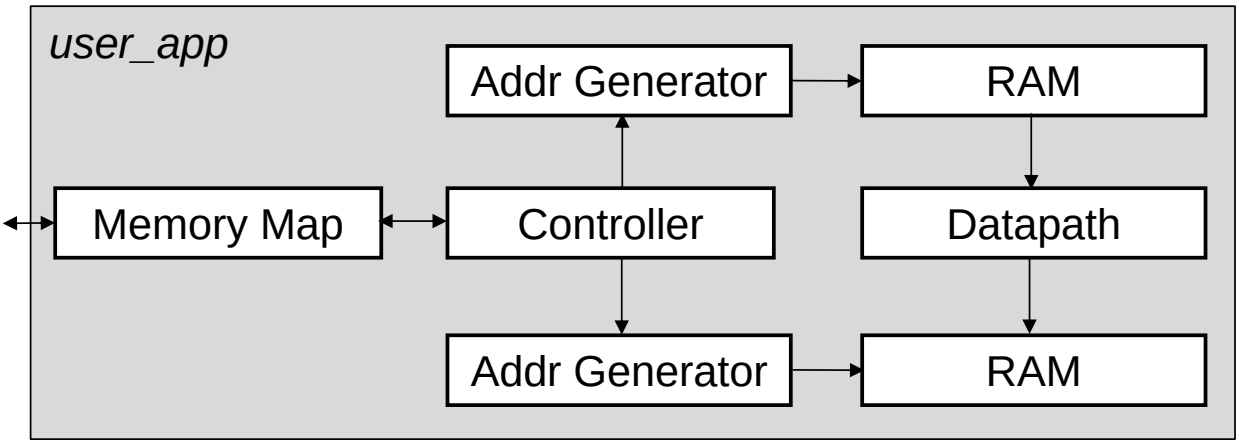
```
for(i=0,j=0; j < OUTPUT_SIZE; i += 4, j++) {

  a[j] = b[i]*b[i+1] + b[i+2]*b[i+3];
}
```

3) Implement the following datapath:



As shown, the datapath has 4 8-bit inputs which connect to two multipliers followed by an adder. The green boxes are all registers of the specified widths. You can create the datapath entity however you want, but it must synthesize to the exact structure shown. Therefore, I would recommend a structural description of registers, multipliers, and an adder. The multipliers should generate a 16-bit output and the adder should produce a 17-bit output to prevent overflow.

4) Implement the rest of the circuit:



The overall structure of the circuit is shown above. You will use a memory map similar to the previous lab. However, for this lab, the memory map also handles transferring data to and from block RAMs. This code is already provided for you. However, you must create all other components.

The controller works similarly to previous labs. It will initially wait for a go signal to be asserted from software via the memory map. When go is asserted, the controller will read from a size register (not shown) that is also specified from software using the memory map, and will then enable the input address generator which will produce the corresponding number of addresses needed to produce the input stream of the specified size from the top RAM. The controller will also enable the output address generator used for storing outputs from the datapath into the bottom RAM.

The address generators in this lab are essentially counters that count from a specified starting address (in this case, from address 0) for the number of addresses specified by *size*. The address generator's output (i.e., the current address) should connect to each RAM. The address generator will likely also include control signals for reading or writing to RAM, although there are numerous different implementations.

The datapath should be implemented as shown earlier. However, each register in the datapath has an enable signal (not shown) that controls stalls. There are numerous ways to control these enable signals. One potential way is to add a control signal from the controller that enables the datapath at the appropriate time. With this approach, the controller would start the input address generator, wait a certain number of cycles, then

enable the datapath, and then enable the output address generator at the exact cycle that would store the first datapath output. Although this approach works for this lab, it does not work well for more complex circuits where latencies are not known or where latencies change. A more flexible way of enabling the datapath is to include an extra flip-flop at each level of the pipeline that represents whether or not the data at that level is valid. With this approach, you wouldn't ever stall the datapath, but would instead change the output address generator to ignore outputs where the valid bit is 0. Either approach is acceptable for this lab.

The RAMs are implemented using provided code that the synthesis tool will infer as a block RAM. To enable communication with your custom block RAMs, I have included code in the memory map entity that demonstrates this functionality. I encourage you to understand the provided code because you will need it for future labs.

5) Simulate your user_app entity with the provided testbench and fix any errors. Like previous labs, this testbench uses different timings than actual on-board execution, so you might want to expand it. You also should change the size of the test in the testbench.
6) Create a Vivado project, add the accelerator IP as an AXI peripheral, and generate a bitfile. Rename the bitfile to lab4.bit. Copy the bitfile to your project directory on reconfig.ece.ufl.edu.

Part 2 – Software

To communicate with the custom circuit, you will again need software that transfers data to the input block RAM, sets the input size, sets the go signal, waits for the done signal, and then reads the outputs from the output block RAM. For this lab, I have provided all software code to demonstrate how to communicate with the block RAMs through the memory map. Make sure you understand the provided code.

IMPORTANT: You can change the software code for testing, but you do not need to submit it. The grader will use the provided software code.

1) Upload the provided software code to reconfig.ece.ufl.edu
2) Compile the software code by running make
3) Execute your design with:

```
zed_schedule ./zed_app lab4.bit > test.txt
```

4) Check the output for errors using (you might first need to do chmod +x grade.pl):

```
./grade.pl test.txt
```

SUBMISSION INSTRUCTIONS (One submission per group)

Make sure all group member names are at the top of every file!

Create a directory with your server account name. Give it the following structure:

```
server_account_name/
        readme.txt              // Group members, anything that the grader needs to
                                // be aware of
        lab4.bit                // DON'T FORGET!!!!
        accelerator_1.0/        // IP core from repository with this exact name
                                // Make sure all VHDL is included
```

Zip the entire directory and submit the server_account_name.zip file.

COMMON PROBLEMS

- You may receive the following warning in ModelSim: "Case choice must be a locally static expression". Although in general you should avoid this warning, it is safe to ignore for this lab. The warning is caused by the use of a function when defining the constants used in the when statements. Most VHDL tools now ignore this issue.