Introduction:

Convolution is a common operation in digital signal processing. In this project, you will be creating a custom convolution circuit implemented on the Zedboard that exploits a significant amount of parallelism to improve performance compared to a microprocessor.

Although it is outside of the scope of the project instructions to give a detailed description of convolution (use google), I have included the pseudocode below:

```
for (i=0; i < outputSize; i++) {
    y[i] = 0;
    for (j=0; j < kernelSize; j++) {
      y[i] += x[i - j] * h[j];
    }
}
```

Convolution takes as input a signal (shown as the x array) and a kernel (shown as the h array). The output is another signal (y array), where each element of the output signal is the sum of the products formed by multiplying all the elements of the kernel with appropriate elements of the input signal. Note that this pseudocode is not complete. Specifically, the x array will be accessed outside of its bounds both at the beginning of execution where i-j is negative and towards the end of execution, where i-j is larger than the x array (the output size is sum of the input size and the kernel minus 1). Also, you will be using 16-bit unsigned integer operations, which need to be "clipped" to the maximum possible 16-bit value in case of overflow. See the provided C++ code for a correct software implementation.

The FPGA implementation will store the input signal in DRAM (currently an emulated DRAM implemented in block RAM), and will read in a kernel (up to 128 elements) through the memory map. The FPGA will then execute like previous labs, using a go and size input from the memory map, while writing all results to another DRAM. Your datapath will fully unroll the inner loop, and then pipeline the outer loop. Due to resource limitations of the FPGA, the maximum kernel size will be limited to 128 elements. However, your circuit will need support all kernel sizes between 1 and 128.

EDGE INSTRUCTIONS: EDGES students must do parts 2-4. All non-EDGE students must do the entire project.

Part 1 – DRAM DMA Interface (EDGE students can use my solution, or can do this for extra credit)

For this project, you will learn how to create a pipeline that streams data to/from external DRAM. However, because the current board template doesn't support external memory accesses yet, I have instead made an emulated DRAM (see dram_model.vhd) that is implemented on block RAMs, but exhibits many of the characteristics of DRAM. The emulated DRAM includes a 10-cycle read latency, and also has refresh periods where the memory should not be accessed.

The main challenge of interfacing with the DRAMs is dealing with both control and data signals that cross clock domains. The DRAMs (and hardware abstraction layer) run on a 100 MHz clock (**FCLK_CLK0**), while the user application (within user_app.vhd) will run at 70 MHz. Use the exact same clocking wizard setup from lab 5.

External memories are generally accessed through a DMA interface where the user specifies the starting address and the size of the transfer. In this part of the project you will be creating a DMA entity for reading from DRAM into your convolution pipeline (dram_rd.vhd). Both the dram_rd.vhd and dram_wr.vhd are provided to you as pre-synthesized solutions. You do not need to implement dram_wr, but you can for extra credit.

For the dram_rd entity, you will need to create an address generator for the DRAM. This will look a lot like your block RAM address generator from earlier labs. Note that the address generator should run on the same clock as the DRAM (FCLK_CLK0), which within my provided solution is call dram_clk.

Next, you will need to integrate the DMA interface with the address generators. At the very least, you will likely have a go signal, a done signal, a starting address signal, and a size signal between the interface and address generator. Because the DMA interface is in a different clock domain, *all of these signals must be synchronized*. I strongly recommend using a handshake synchronizer between the interface and address generator to ensure that the signals are stable before the address generator uses them. This is critical for reliable functionality. Without synchronization, the DRAM might occasionally work, but there will be non-deterministic behavior.

In addition to the previous signals, you will obviously need to also handle transferring data from memory into the DMA interface's data output that will be used by the pipeline. To implement the transfer of data across domains, use a FIFO with different read and write clocks. Also, the DRAM delivers 32 bits at a time, but your application works on 16-bit data. Therefore, your FIFO for dram_rd should have a write width of 32 bits and a read width of 16 bits. Note that there are other ways of implementing this data transfer, but I would strongly suggest this one due to the buffering techniques that will be discussed later. For the input FIFO, make sure to use a programmable full flag that leaves at least 10 entries for outstanding read requests. Also, I'd recommend using the "first-word fall through" setting to simplify the control logic.

It is critically important to realize that the rst signal on the Vivado FIFO IP is synchronized to the write domain. So, make sure you use the appropriate reset. The dram_rd entity has two reset inputs: one for each domain.

Interfacing with DRAMs is not trivial, so I have provided a sample program that you can use to test your design. The provided code uses a datapath that simply passes data through unchanged from ram0 to ram1, which effectively allows you to test transferring data from one DRAM to another through the FPGA. .

To help you get started, I have provided pre-synthesized versions of both DMA entities (see the edif files in the src/ directory). You use these cores by including it in your project and instantiating it in a VHDL port map, which is already done for you. Unfortunately, you can't simulate edif files, so I have also included my solution in an encrypted form. See the *.vhdp files. If you are wondering why I provided edif files and the encrypted solution, for some reason Vivado wasn't synthesizing my encrypted files, but simulates them just fine.

Before writing any of your own code, create a base project that is identical to lab 5, but for the peripheral, use the provided dram_test IP. Edit the IP in the IP packager. I have provided a basic testbench (wrapper_tb.vhd) to assist you, but be aware that you should make significant changes for testing purposes. Run a behavioral simulation and ensure that the tests report no errors. *In most cases, you will see errors*. **These errors occur because for some reason Vivado keeps removing the dma_fifo IP from the peripheral, which is needed by my simulation code. To fix this, manually add the dma_fifo.xci file as a design source, then re-simulate and it should work.**

Next, merge changes and repackage the IP. Then, in the base project, create a bitstream. Save that bitstream and upload it to the server along with the provided sw/ folder.

You do not need to make any changes to the software. On the server, type “make” and then run the application in the usual way:

zed_schedule ./zed_app bitfile

This should run through a large number of tests. If errors are reported, something when wrong in the earlier steps. If it is successful, you are ready to start replacing dram_rd with your own code.

Re-open the peripheral project and remove dram_rd.edif from the design sources and dram_rd.vhdp from the simulation sources. Replace them with your own dram_rd.vhd file and start implementing your own functionality. Use the provided wrapper_tb testbench, but make sure to make modifications to test specification functionality. The provided tests were chosen somewhat randomly. Keep in mind, you might have to re-add dma_fifo.xci as a design source every time you open the IP project.

Once you are confident that your design simulates correctly, I would recommend first testing it with a base project that uses a single clock (just connect FCLK_CLK0 to both clk inputs on the peripheral). This allows you to debug on the board without having to worry about metastability.

After you get the dram_test software running without an errors using a single clock, go back to the setup from lab 5 with two clocks and re-test it. There is a good chance there will now be errors caused by metastability.

When your design works perfectly, you should see the following:

```
Starting job "./zed_app design_1_wrapper.bit" on board 192.168.1.107:
Programming FPGA....SUCCESS
Testing transfers to/from address 0....SUCCESS
Testing max transfer size....SUCCESS
Testing random sizes and addresses....SUCCESS
```

The DMA code is by far the most complicated part of the project. You will almost certainly run into bugs that are difficult to understand. It is critically important that you implement modifcations to the testbench to figure out problems. Remember, that with the exception of metastability, the first step in debugging issues on the board is to recreate the issue in simulation.

Similarly, feel free to modify the C++ code to print out useful debugging information. You can’t change the C++ code to make your peripheral work, but you can change it to debug.

Part 2 – Convolution pipeline

Make a new peripheral in Vivado called convolve_1.0. Copy all the source and IP from the dram_test example. The easiest way to do this is to simply copy the IP folder, open it in the IP packager, and then rename the IP from dram_test to convolve. In the remaining parts, you will be modifying the convolve IP.

Due to time constraints, I will be providing most of the solution to the pipeline itself (see mult_add_tree.vhd). However, you are still welcome to do this as extra credit. If you use my code, make sure to instantiate the unsigned_arch architecture.

The pipeline for convolution is large, but conceptually simple. In this project, we will limit the maximum kernel size to 128. To support this kernel size, the first row of the pipeline consists of 128 16-bit multipliers, each of which will multiply corresponding elements from the signal and kernel. After the first row, an adder tree is used to add all the products together. The output of

the adder tree (i.e., final sum) defines a single output element, which is the output of the datapath.

If you decide to implement the pipeline yourself, there are several things to be aware of. First, the pipeline should have a register after every multiplier or adder. Second, the datapath takes 16-bit inputs and produces 16-bit outputs, which means you don't need to increase the width of operations as you go deeper into the datapath. However, you will need to implement saturation (i.e., clipping, clamping, etc.), which means that if at any point in the datapath the result of an operation exceeds 16 bits, the output of the operation should be all 1's (0xffff). If you don't implement this clipping, the results will wrap back around to zero, which will not be correct. There are two ways to do this: 1) implement saturation within each adder/multiplier, or 2) perform the multiply-add tree while storing all overflow bits, and then implement saturation on the pipeline out. Unless you want to modify the code I provided, I recommend the 2nd method.

The entire pipeline consists of 128 multipliers, 127 adders, and a lot of pipeline registers. If you implement your own pipeline, I highly recommend using VHDL generate statements and arrays, otherwise your code will be huge. You can actually use recursion to create very concise structural architectures, which is demonstrated by my provided code.

Part 3 –Signal Buffer

You might have noticed that there is potential problem in the first two parts of the project. The DRAM only delivers 32 bits at a time, and the DMA interface only outputs 16 bits at a time, but the pipeline needs 128*16 bits every cycle to avoid stalls. Fortunately, there is significant overlap between iterations. In fact, convolution is a sliding-window algorithm where the window moves by one element each iteration. Therefore, we can exploit the overlap between iterations to reuse data and improve bandwidth. There will be a class lecture that discusses the exact details.

To enable data reuse, you will need to create a simple buffer that generates 128-element signal windows for the pipeline. Because the window only differs by a single element each iteration, the buffer can be implemented as a large shift register that shifts in 16 bits at a time.

You are free to implement the buffer however you like, but I'd recommend using a FIFO-like interface that specifies if the buffer is empty/full. With this interface, the pipeline can read whenever the buffer isn't empty and you can write data into the buffer (from the ram0 read DMA interface) whenever the buffer isn't full. The data input to the FIFO should be 16 bits, and the output should be an array of 128 16-bit elements. The easiest way of handling this output is by creating an array type that is defined in a package (e.g., user_pkg.vhd), so that you can use the array in the port definition. Assuming you implement everything correctly, this buffer will be capable of delivering a 128-element window to the pipeline every cycle (assuming data is available from the DRAM). If your speedup is significantly less than what is demonstrated in class, then the problem is likely that this buffer does not provide a new window every cycle.

Part 4 –Kernel Buffer

In addition to buffering the signal as it is read from memory, you need some way of storing and accessing the entire kernel. Although we could potentially read the kernel from memory, because I have kept things simple by limiting the kernel size to 128 elements, you can transfer the kernel into registers in the FPGA using the memory map. In the provided memory map for convolution, there is an address, KERNEL_DATA_ADDR, that is used for these transfers. You should use another shift register that consists of 128 16-bit elements, which shifts in 16 bits every time that the memory map writes data to an address that corresponds to the kernel. After 128 memory map transfers, the entire kernel should be loaded. Note that this functionality is a subset of part 3, so ideally you can reuse the same buffer entity. It would probably be a good

idea to check if the kernel is actually loaded during testing. To support this, I included a KERNEL_LOADED_ADDR in the memory map that you can use to read the status of the kernel buffer. This is completely optional, but provides information for debugging if you want to use it.

Other tasks

In addition to the previously discussed components, you might also need a simple controller, glue logic, etc. I have provided the memory map (memory_map.vhd in the convolve directory), but you might need to modify it based on your implementation of other components. I would highly recommend basing your design on the provided dram_test code. Notice I have removed the starting addresses in the convolution memory map because we can assume that the signal and output both start at address 0. **Make sure you use the provided memory map in the convolve directory because the dram_test memory map is not compatible with the convolve software.**

After you have implemented all components, create your own testbench by modifying the provided testbench (wrapper_tb.vhd). IMPORTANT: This testbench is just a basic template that was quickly adapted from the dram_test testbench. It does not check any output values for correctness. It always uses the maximum kernel size. You must extend it to match the behavior of software. At a bare minimum, you should pad the kernel to test actual kernel sizes that are smaller than the maximum size. The vast majority of the software tests will do this, so your testbench should match that functionality.

When you are confident that it is working, create a bitfile. Test your implementation with the provided C++ code on the class server. You cannot modify the C++ code to fix problems in the VHDL, but you are welcome to change it for debugging and testing purposes.

The C++ code does a variety of tests ranging from simple tests that are likely to work up to stress tests that use the largest possible signals with random values. If your code works properly, you should see an output like this:

```
Starting job "./zed_app design_1_wrapper.bit" on board 192.168.1.102:
Programming FPGA....Testing small signal/kernel with all 0s...
Percent correct = 100
Speedup = 0.0159574

Testing small signal/kernel with all 1s...
Percent correct = 100
Speedup = 0.0205479

Testing small signal/kernel with random values (no clipping)...
Percent correct = 100
Speedup = 0.0205479

Testing medium signal/kernel with random values (no clipping)...
Percent correct = 100
Speedup = 3.07368

Testing big signal/kernel with random values (no clipping)...
Percent correct = 100
Speedup = 15.9339

Testing small signal/kernel with random values...
Percent correct = 100
```

```
Speedup = 0.0196078

Testing medium signal/kernel with random values...
Percent correct = 100
Speedup = 2.80223

Testing big signal/kernel with random values...
Percent correct = 100
Speedup = 14.5488

TOTAL SCORE = 100 out of 100
```

Debugging suggestions: I would highly recommend initially testing your design with a reduced maximum kernel size (e.g., 3 instead of 128). You can change this by modifying C_KERNEL_SIZE in user_pkg.vhd. This will make it much easier to mentally follow what is going on in a simulation. Then, start with simple tests (a kernel where each element is 1, a signal where each element is 1). Then, slowly start making it more complex until you eventually use random values for the signal and kernel.

Probably >90% of your time for this part of the project should be spent making the testbench as good as you can, and then analyzing waveforms.

Extra Credit:

If you finish early, there are quite a few options for extra credit. In the current implementation, you should notice that all padding of the signal (i.e., handling the cases where the bounds of the signal are exceeded) and kernel (i.e., handling kernels less than 128 elements) is performed in software. In other words, 0's are added to the beginning and end of the signal, and kernels less than 128 elements have 0's added to the beginning. Padding in software works, but is potentially slow, especially for large signals. One good extension for extra credit is to implement the padding in the FPGA itself. Note that this will also require changes to the software code.

If you really want to impress me, you can also extend the FPGA implementation to handle arbitrary kernel sizes. I'll warn you that this is not trivial, and I won't give you the exact techniques required, but here are a few hints. To implement an arbitrarily sized kernel, you must perform the convolution multiple times, each time with a different 128-element segment of the kernel. The tricky part is making sure the padding of the signal is done correctly each time. In addition, the output of each iteration is a partial result, which must be accumulated with multiple future iterations to get the correct answer. Do not attempt this step until you have finished everything else.

Another possibility would be to convert your implementation to use floating point instead of integer operations. You will not be able to fit as many operations on the device, but if you implemented your VHDL in a good way, this should not require many changes to your code.

You could also create a frequency-domain implementation that uses an FFT and IFFT. I won't explain this implementation, but there is plenty of information on google.

I would also like to see the code demonstrated on a real example. The use of 16-bit signals and kernels makes this project perfect for testing 16-bit audio. Note that you might need to make some changes to the pipeline to support different types. For example, audio usually uses signed values.

All extra credit should be turned in using a separate directory so there is no confusion when grading the original project.

Report Instructions: IMPORTANT

If your project is not 100% complete, it is up to you to show me what you have completed. Therefore, you should include a report that shows testbench simulations, explanations, etc. that demonstrate that certain parts of the project are correct. Do not simply say that "we finished the DRAM interface", or you will not get credit. However, if you show detailed waveform simulations and correct output from the provided test application, then I will be convinced. If you have fully completed the project, the report can simply be a brief description of your implementation, a summary of any problems you encountered, and how you fixed those problems. If you did any extra credit, it should be described in the report.

SUBMISSION INSTRUCTIONS (One submission per group)

Make sure the names of all group members are at the top of every file that you create and/or modify.

Create a directory with named after your UFID. Use the following structure:

```
ufid/
    readme.txt              // Group members, anything that the grader needs to
                            // be aware of
    report.doc or pdf       // detailed report explaining what works (with
                            // convincing evidence)
    dram_test.bit           // Only include if working in a group.
    dram_test_1.0/          // IP core extended with your code (if in a group)
    convolve.bit            // Bitfile for your convolve application
    convolve_1.0/           // Convolution IP core
    extra_credit/           // If applicable, put any extra credit files in here.
                            // Include a separate report in this directory that
                            // explains the extra credit.
```

Zip the entire directory and submit the zip file. Note that you do not have to submit any C++ code, unless you did something for extra credit.