Introduction:

Convolution is a common operation in digital signal processing. In this project, you will be creating a custom circuit implemented on the Nallatech board 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 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 SRAM (up to 4 megabytes), and will read in a kernel (up to 96 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 SRAM. Your datapath will fully unroll the inner loop, and then pipeline the outer loop. Due to resource limitations of the FPGA, the kernel size will be limited to 96 elements, which is the number of DSP units on the device.

Part 1 – SRAM Interface (solution provided to those not working in groups)

For this project, you will be using the SRAMs on the Nallatech board in order to support larger signals (4 megabytes) that will not fit in block RAM. The details of the SRAMs are discussed in the Dimetalk Reference Guide on pages 104-112.

The main challenge of interfacing with the SRAMs is dealing with both control and data signals that cross clock domains. The SRAMs run on a 200 MHz clock, while the rest of your circuit will run on clkA. Even if you run clkA at 200 MHz, you will still have metastability problems.

First, you will need to create an address generator for the SRAM. This will look a lot like your block RAM address generator, except that SRAM addresses need to be incremented by 2 (the addresses should only be even). Note that the address generator should run on the same clock as the SRAM (200 MHz).

Next, you will need to interface your controller 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 controller and address generator. Because the controller is in a different clock domain, *all of these signals must be synchronized*. I strongly recommend using a handshake synchronizer between the controller and address generator to ensure that all the signals are

stable before the address generator uses them. This is critical for reliable functionality. Without synchronization, the SRAM 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 datapath and vice versa. To implement the transfer of data efficiently, use a FIFO with different read and write clocks. Also, the SRAM delivers 64 bits at a time, but your datapath works on 16-bit data. Therefore, your input FIFO should have a write width of 64 bits and a read width of 16 bits. The output FIFO should be the opposite. 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 16 entries for outstanding read requests. The output FIFO does not require a programmable full flag. For both FIFOs, I'd recommend using the "first-word fall through" setting.

Interfacing with the SRAMs 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, which effectively allows you to test transferring data from one SRAM to another through the FPGA. In the top-level file (sram_test_h101.vhd), your SRAM interfaces should be implemented in the sram_rd and sram_wr entities.

For those working by themselves, I have also provided pre-synthesized implementations (ngc files) for sram_rd and sram_wr. You use these just like cores from Coregen, by either having the ngc files in the same directory as your ISE project, or by including them as support files in Dimetalk. Unfortunately, you can't simulate ngc files, so I have also included simulatable models (sram_rd_sim and sram_wr_sim) for each of these entities. These models define the sram_rd and sram_wr entities, so if you use your own implementations, make sure to remove these from the ISE project. Do not synthesize the entities defined in the _sim.vhd files! They will not work. Also, do not include the _sim.vhd files as support files in Dimetalk.

I have provided a testbench to assist you, but be aware it does not use assertions. You must verify correctness yourself.

Next, open the provided Dimetalk project. You will likely receive a warning about the sram_test_h101 component not being found. This occurs because Dimetalk does not use relative addresses, which means if you don't have all the VHDL files in the same directory I did, it won't be able to find the files. To fix this, first make a mental note of how the components are connected to sram_test_h101. Next, delete the sram_test_h101 component from the schematic. Replace it by going to View->Library Manager->Add to Library->VHDL Component. Select the top-level file (sram_test_h101.vhd). Make sure to add all the vhdl and ngc files as support files (assuming you are using my provided implementation), with the exception of sram_rd_sim and sram_wr_sim. If you are using your own implementation, include your sram_rd and sram_wr vhd files along with whatever ngc files you used for your FIFOs. Make sure to not include my ngc files and your sram_rd/wr vhdl at the same time. Next, add the sram_test_h101 to the schematic, and reconnect everything. Finally, create the bitfile and execute the provided C++ code, which will test your SRAM implementation and print any errors that are found.

There are quite a few files provided in the sample code, which are explained in the readme.txt file. Make sure to read this file before asking any questions about the purpose or location of a file. The readme file also includes an important description of files that are required by the testbench.

Part 2 – Datapath

The datapath for convolution is large, but conceptually simple. The amount of unrolling is limited by the 96 DSPs in the device, which are needed for multiplication. The first row of the datapath consists of 96 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.

You are free to implement the datapath in numerous ways, but there are several things to be aware of. First, the datapath should be pipelined so that there is a register between 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 (e.g., clipping of an audio signal). In other words, 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.

The entire datapath consists of 96 multipliers, 95 adders, and a lot of pipeline registers. I highly recommend using VHDL generate statements and arrays, otherwise your code will be huge. There is actually another trick you can do for the adder tree using recursive structural descriptions that allows you to create a tree of any size based on generics. However, I don't recommend this recursive technique initially, unless you are very comfortable with recursion. Also, I have not tested the recursion with the version of ISE you are using, so it might not even be supported.

Part 3 –Signal Buffer

You might have noticed that there is potential problem in the first two parts of the project. The SRAM only delivers 64 bits at a time, and the input FIFO only outputs 16 bits at a time, but the datapath needs 96*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.

To enable data reuse, you will need to create a simple buffer that generates 96-element signal windows for the datapath. 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 datapath can read whenever the buffer isn't empty and you can write data into the buffer (from the input SRAM) whenever the buffer isn't full. The data input to the FIFO should be 16 bits, and the output should be an array of 96 16-bit elements. The easiest way of handling this output is by creating an array type that is defined in a package, so that you can use the array in the port definition. Assuming you implement everything correctly, this buffer will be capable of delivering a 96-element window to the datapath every cycle (assuming data is available from the SRAM).

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 96 elements (due to the DSP restriction), you can transfer the kernel into registers in the FPGA using the memory map. Basically, you should use another shift register that consists of 96 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 96 memory map transfers, the entire kernel should be loaded. Note that this functionality is identical to part 3, so ideally you can reuse the same buffer entity.

Other tasks

In addition to the previously discussed components, you will also need a simple controller, glue logic for the memory map, etc. I have provided the glue_logic, but you might need to modify it based on your implementation of other components. I would highly recommend basing your design on the provided sram_test_h101 code.

After you have implemented all components, test your VHDL using the provided testbench (again, there are no assertions so it is up to you to ensure correctness). Note that you will need to modify this testbench to match the name of your top-level entity. I used the name "wrapper", but you probably want something more meaningful, like "convolve_h101". You may also need to modify it to test different input values and sizes. When you are confident that it is working, import your code into the provided Dimetalk project and create a bitfile. Test your implementation with the provided C++ code.

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 96 elements) is performed in software. In other words, 0's are added to the beginning and end of the signal, and kernels less than 96 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 96-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.

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 SRAM interface", or you will not get credit. However, if you show 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.