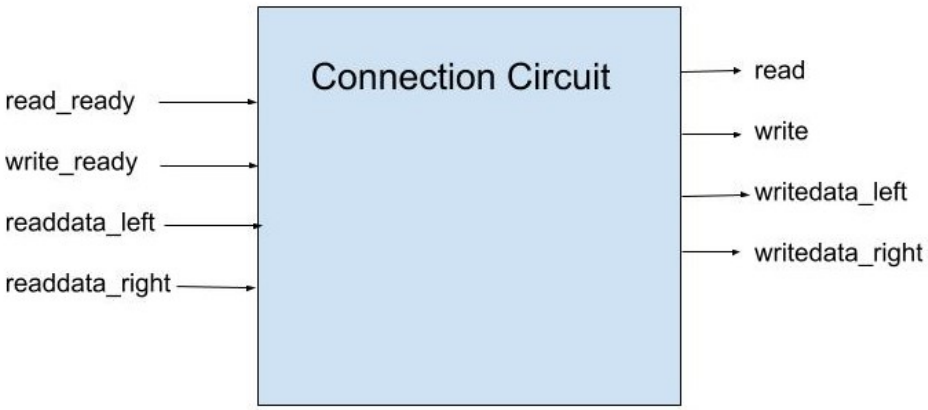
Student Names: Rafael Laya
Student ID Numbers: 1978374

EE/CSE 371: Laboratory #5, Digital Signal Processing

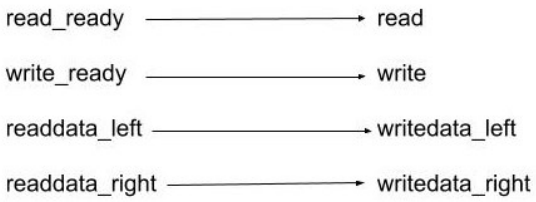
Design Procedure

Task #1

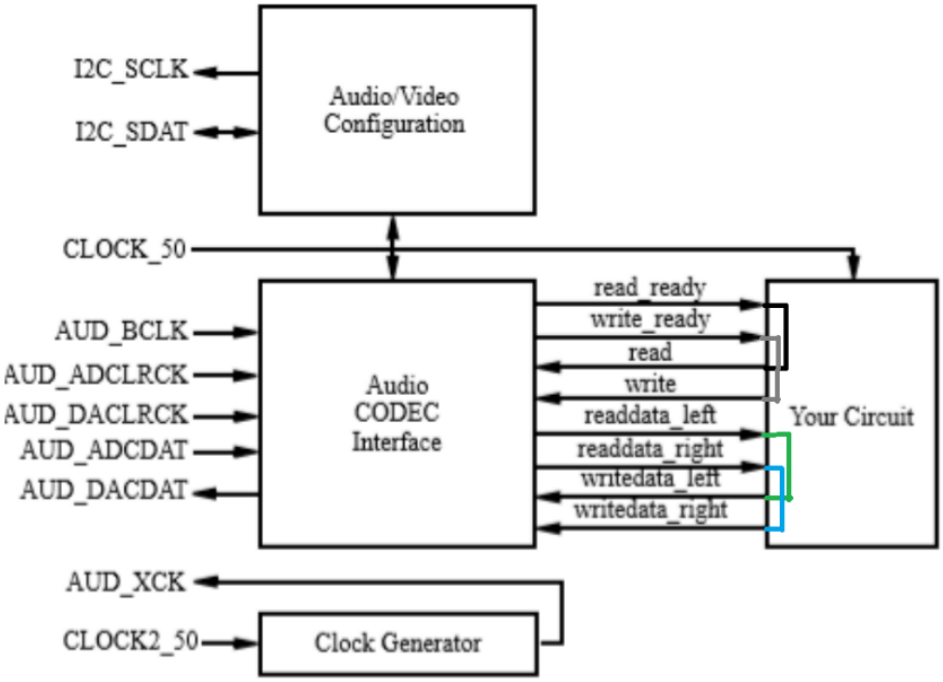
For the first task, the *connection_circuit* module attached as *connection_circuit.sv* makes the connections necessary to play audio using the audio CODEC interface. The block diagram of this circuit is the following:



The internals are really simple, since all that the circuit needs to do is connect each audio channel and to set the read and write signals when its corresponding ready signal is asserted. The FIFO described in the laboratory document then will take care of the rest:

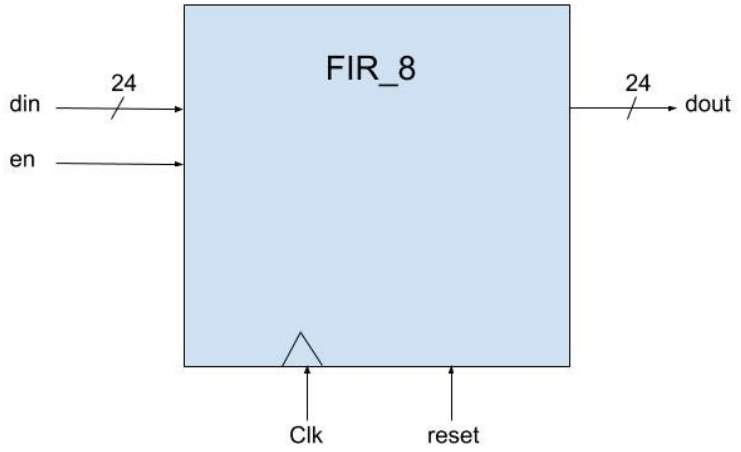


The connections in the top-level module (attached as *part1.sv*) then look as follows:

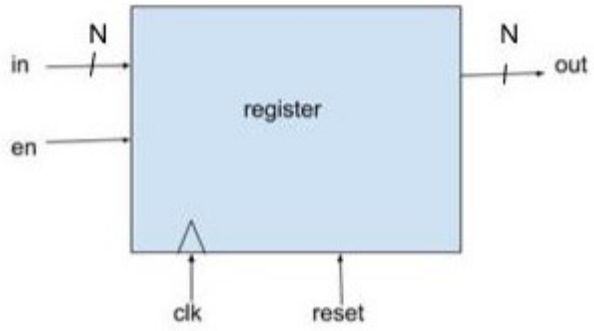


Task #2

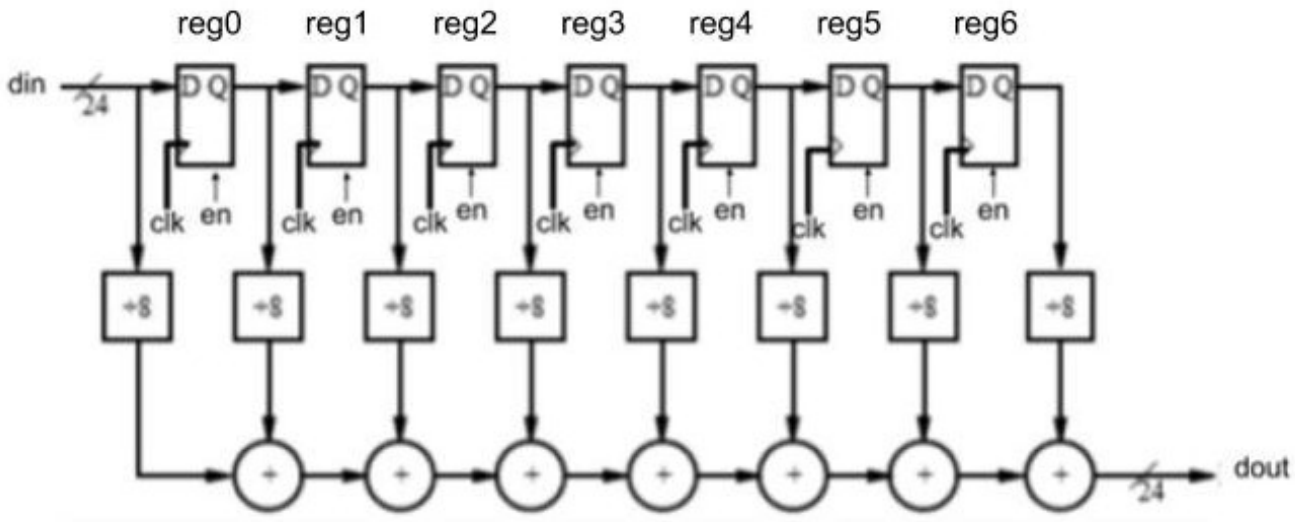
In task two, the module *FIR_8* (attached as *FIR_8.sv*) is developed. The module looks like the following block diagram to the external world. It uses an enable signal *en* since it uses internal registers to produce an enabled shift register (enabled at each time a new sample is read).



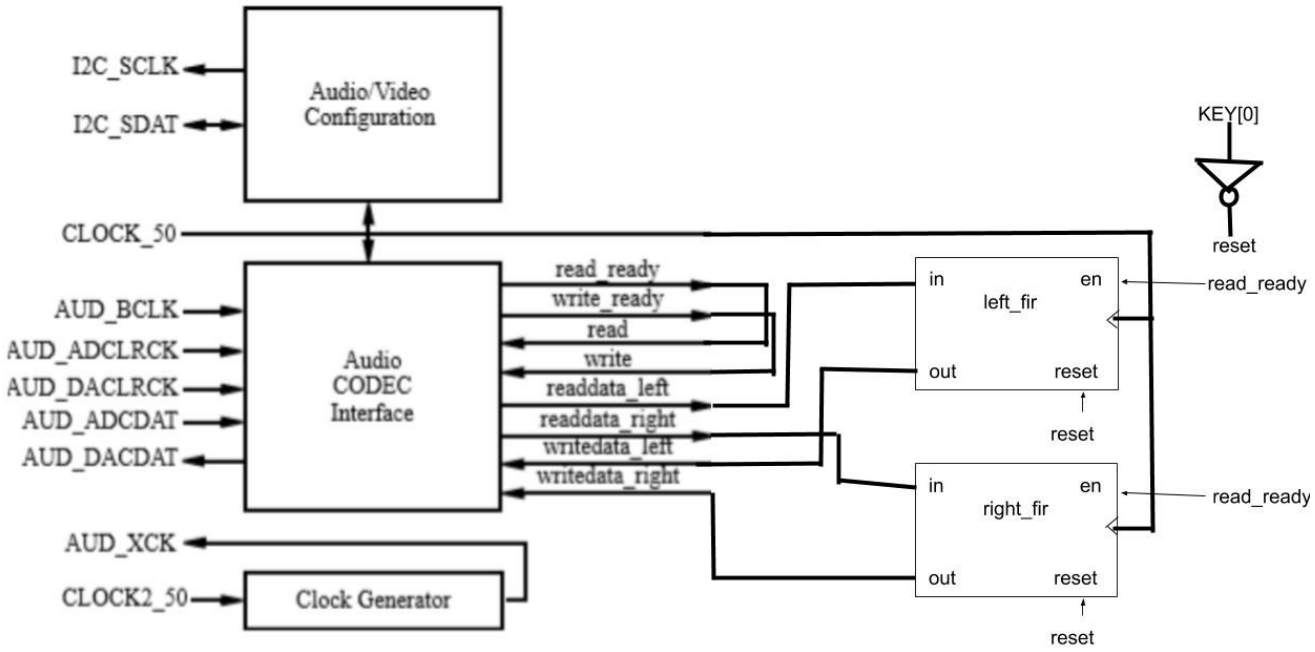
The registers (attached as *register.sv*) are simple enabled registers of parametrized width *N*:



The following picture shows the implementation of *FIR_8*:



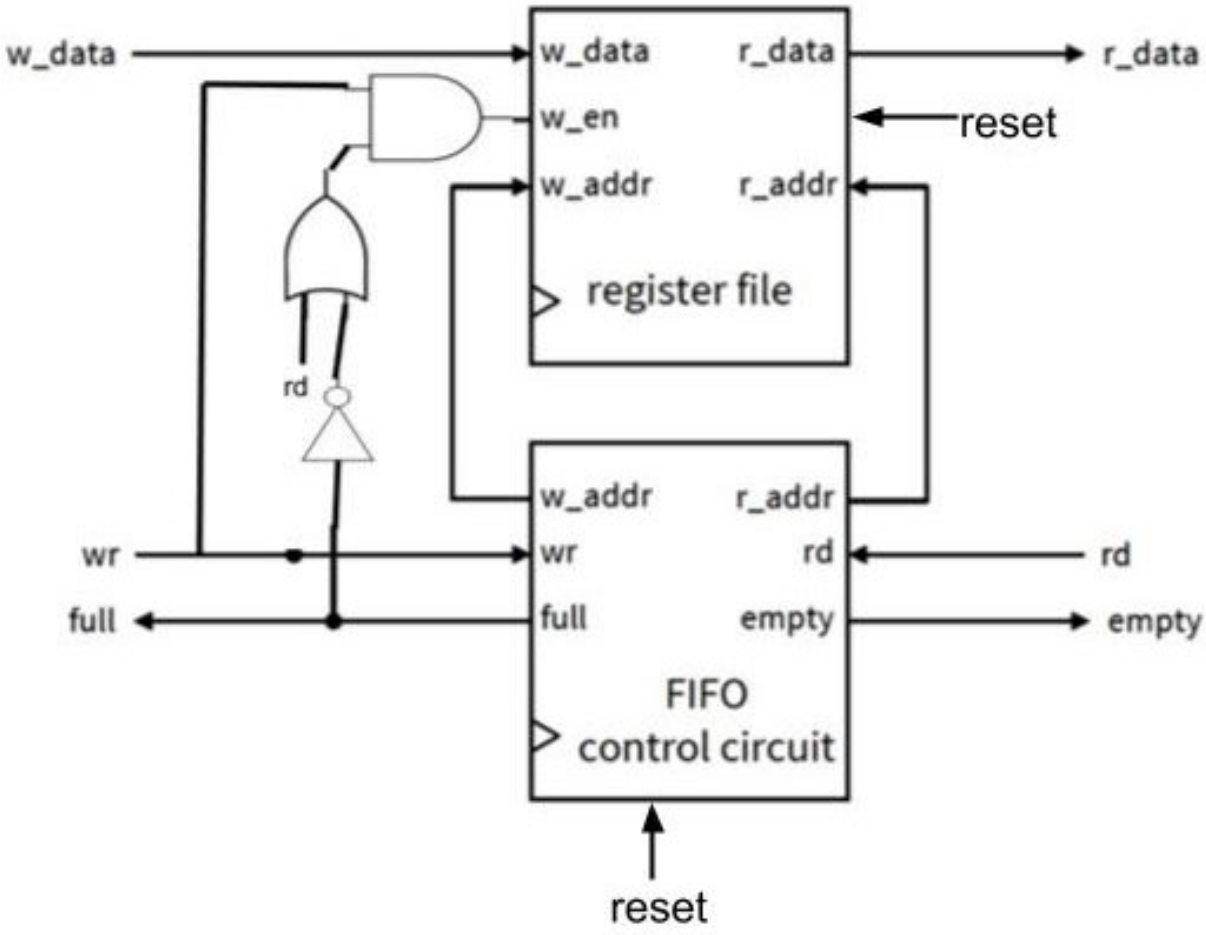
The top-level module for this task takes two *FIR_8* modules, one for each channel (left and right). The top-level module diagram is shown next:



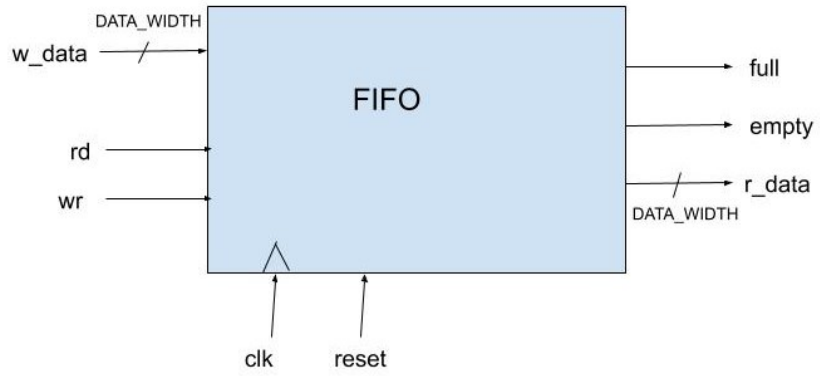
Each FIR is responsible for taking one channel of data and writing to its respective channel through the audio CODEC interface. The code for this module has been attached as *part2.sv*.

Task #3

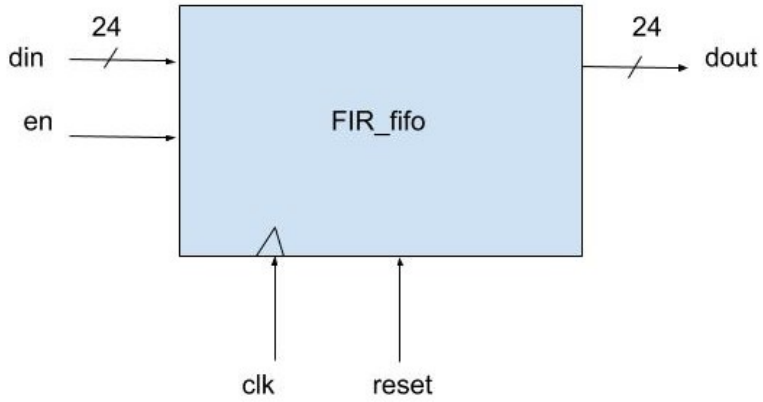
For task 3, first recall the FIFO implementation seen in lecture and in previous homework assignments that uses a register file (*reg_file.sv*) and a control circuit (*fifo_ctrl.sv)*:



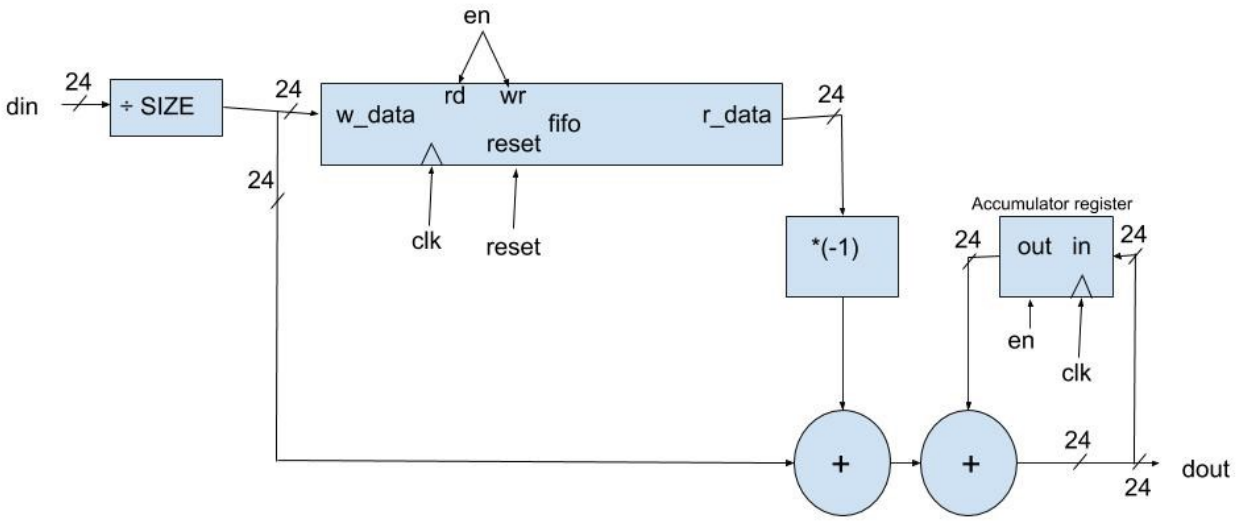
This FIFO has been attached as *fifo.sv*. The module looks like the following block to the external world:



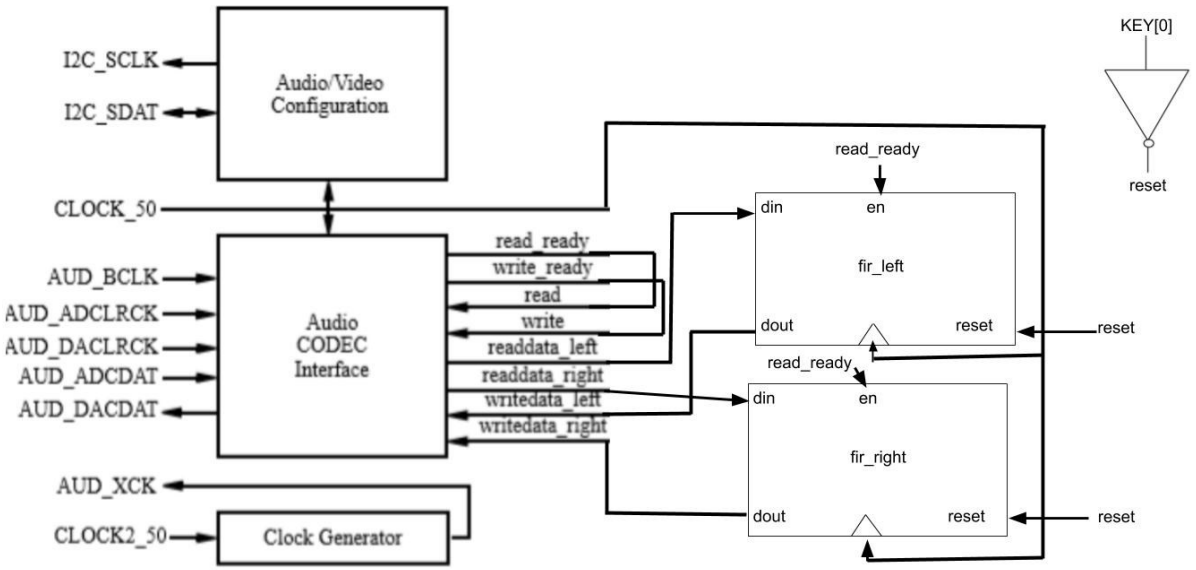
Using this, the goal is to build a FIR filter that has the following inputs and outputs:



The following is an implementation of this FIR:



The FIFO's read and write signal are asserted during each read, which is when a new sample is found and makes the signals propagate. The rest of the logic involves a register that accumulates the result and combinational circuits for performing arithmetic manipulations. This module has been attached as *FIR_Fifo.sv*. Next, the top-level module is shown:



This top-level module (attached as *part3.sv*) uses two *FIR_Fifo* modules, each one is responsible for taking the data from the audio CODEC interface for one channel, and producing the write data for the same interface.

Results

Overview

Task 1 successfully demonstrates a simple connection from the microphone line to the speaker. One can clearly hear an output from the speaker. YouTube videos can be played and listened to through this configuration without any issues.

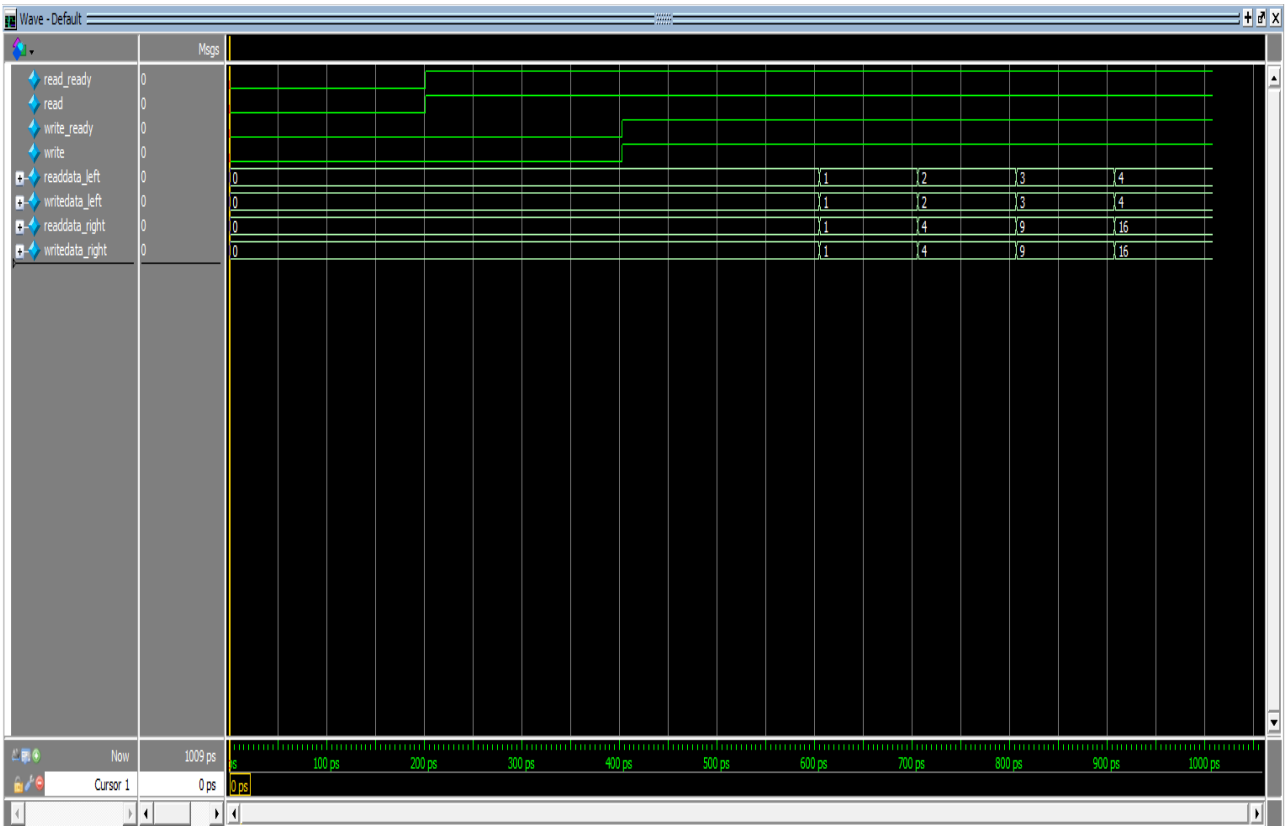
Task 2 successfully improves on the design from task 1 by adding a noise filter using a Finite Impulse Response filter that averages 8 samples to reduce high-frequency noise. One can easily play audio on the microphone line and hear audio from the speaker, and one can notice a small but significant difference in reduced noise which makes voices sound cleaner than task 1.

Task 3 implements another Finite Impulse Response filter that averages the latest N samples to reduce high-frequency noise. After experimenting with different values for N, I noticed that what sounds best to me is N=32 samples. For the fun option, I think N=1024 sounds quite robotic.

Task #1

Since this circuit is only making simple direct connections, the testbench waveforms are easy to read. Each signal in the screenshot is ordered so that the input signal is above the output signal that corresponds to a connection. Therefore, each row that corresponds to an output (rows 2, 4, 6, and 8 when indexing from *read_ready* being row 1) will look like a copy of the signal right above. First is the output in the console that results from the assertions in the testbench:

```
# read_ready == read: TRUE
# read_ready == read: TRUE
# write_ready == write: TRUE
# write_ready == write: TRUE
# readdata_left == writedata_left: TRUE
# readdata_right == writedata_right: TRUE
# readdata_left == writedata_left: TRUE
# readdata_right == writedata_right: TRUE
# readdata_left == writedata_left: TRUE
# readdata_right == writedata_right: TRUE
# readdata_left == writedata_left: TRUE
# readdata_right == writedata_right: TRUE
# readdata_left == writedata_left: TRUE
# readdata_right == writedata_right: TRUE
```



Flow Summary

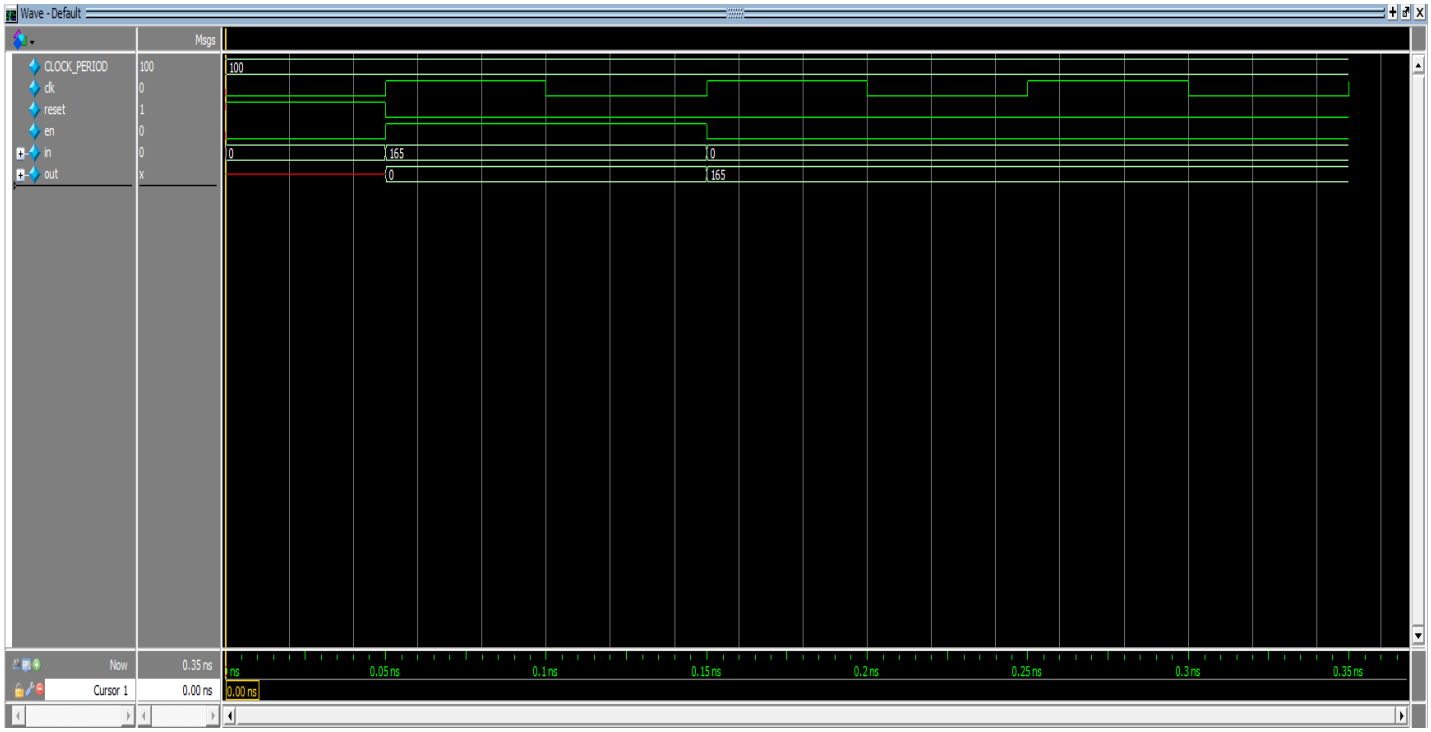
<<Filter>>

Flow Status	Successful - Mon May 18 15:45:15 2020
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	part1
Top-level Entity Name	part1
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	216 / 32,070 (< 1 %)
Total registers	285
Total pins	14 / 457 (3 %)
Total virtual pins	0
Total block memory bits	12,288 / 4,065,280 (< 1 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	1 / 6 (17 %)
Total DLLs	0 / 4 (0 %)

Task #2

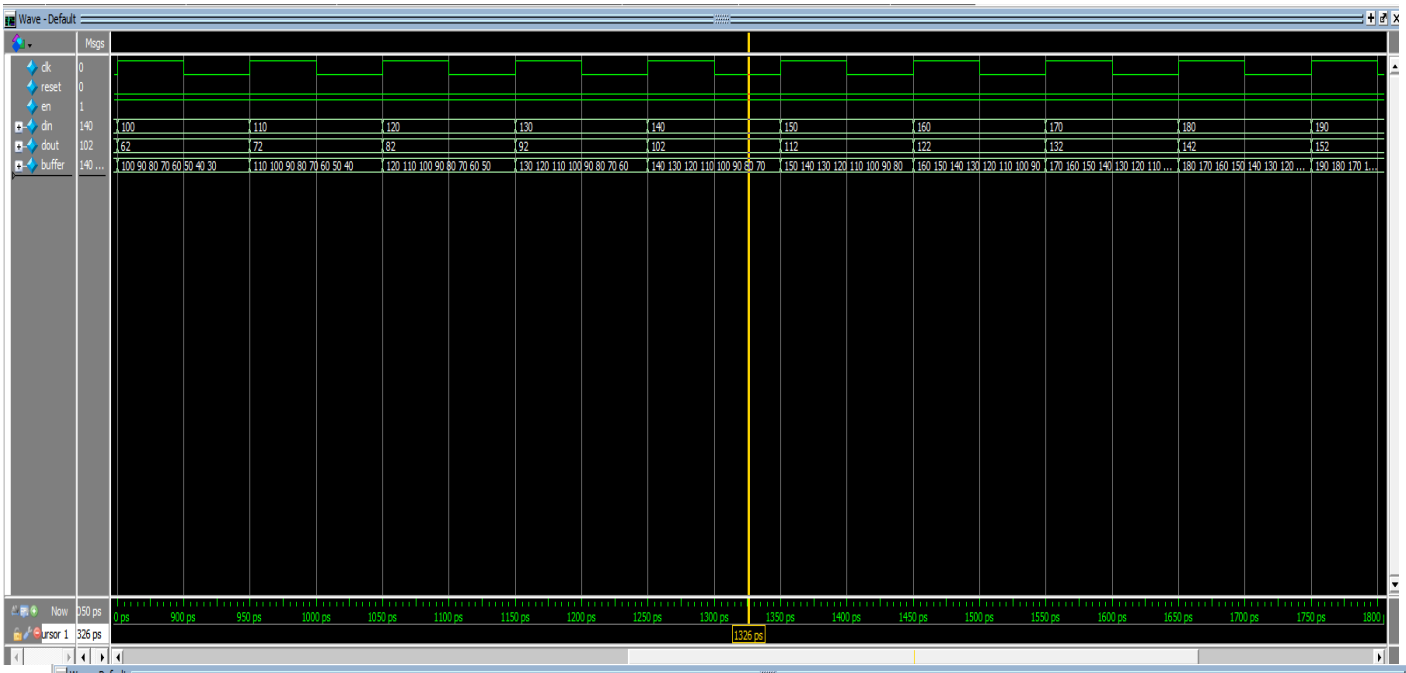
First, the *register* module is tested. The testbench loads (*en* is TRUE) the value 165 and then checks that you can retain a value if *en* is FALSE which is why 165 doesn't change further.

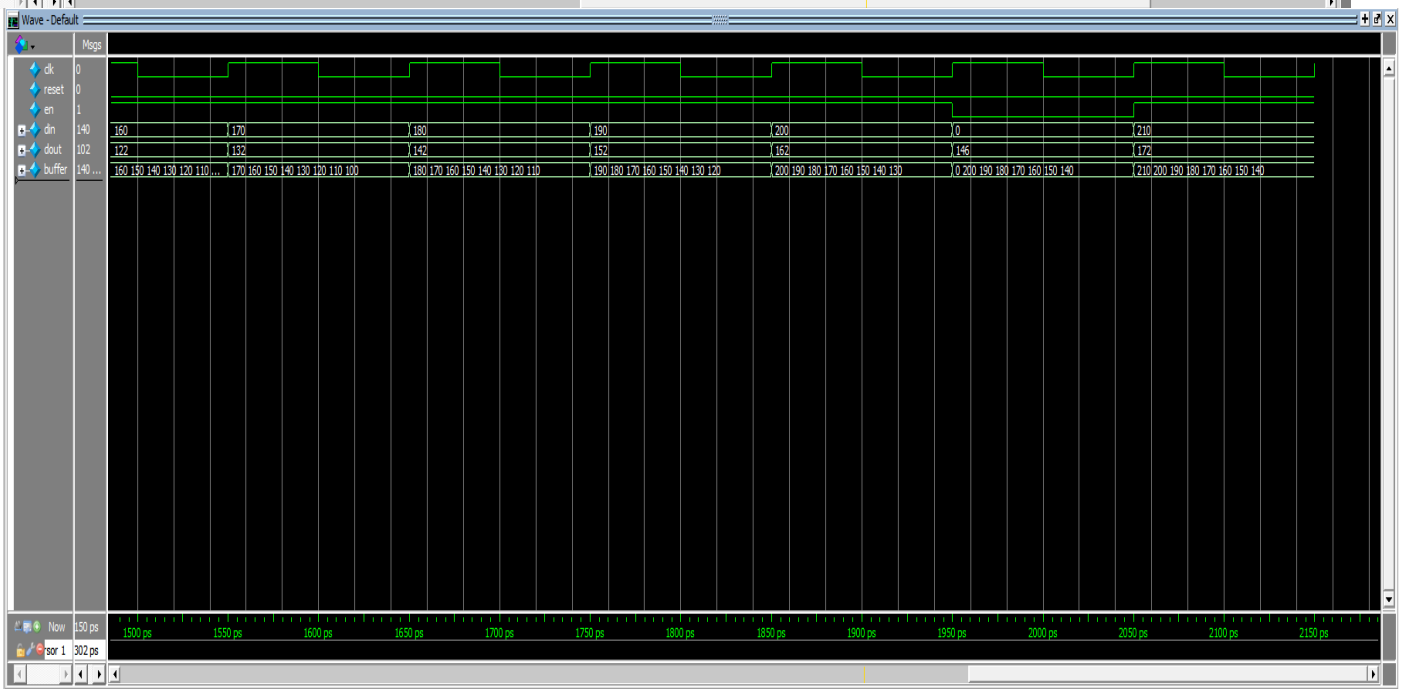
- Testing the register is simple: First it checks that you can load a value (165) and then checks that you retain a value if *en* is FALSE, thus 165 doesn't further change



The *FIR_8* module testbench goes as follows:

1. First starting values are loaded by pushing 30, 40, 50, 60, 70, 80, and 90 into the registers (omitted since the testing occurs after 850ps).
2. When 100 is pushed as input data, the average becomes 62. The average is 62 because we have 30/8+40/8+50/8+60/8+70/8+80/8+90/8+100/8=3+5+6+7+8+10+11+12=62. The real average of these numbers is 65, but the spec shows division performed before the additions which introduces a small truncation error (notice however that the bigger the numbers, the smaller the percent error thus these 24-bit numbers should not be of concern).
3. Then 110 is shifted in, the average becomes 72 as expected.
4. Similarly, the process is continued by pushing every time a number 10 units larger, until we arrive at 200 with an average of 162.
5. Finally, the *en* (enable) signal is set to LOW, and *din* is loaded with 0. Since *en* is LOW, the value is invalid during that clock cycle, and then when *en* becomes HIGH again, the average is computed and the value of *din* that wasn't valid is not shifted into the buffer.



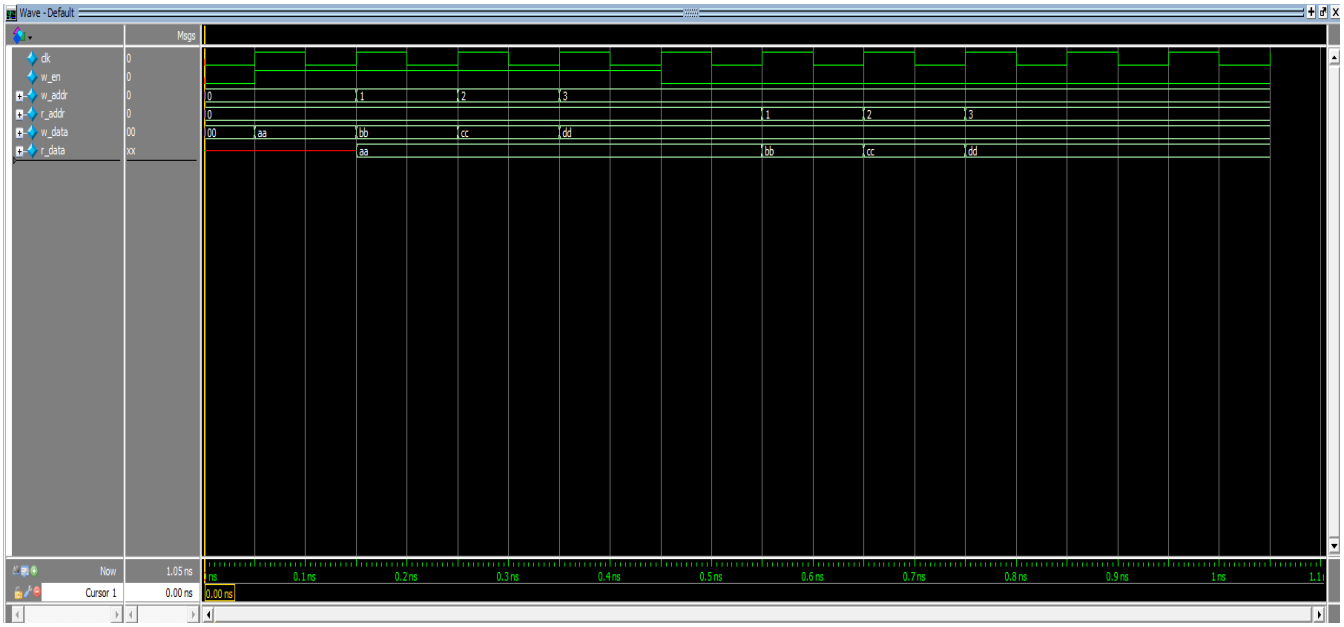



```
# Success. Expected:    62, Got:    62
# Success. Expected:    72, Got:    72
# Success. Expected:    82, Got:    82
# Success. Expected:    92, Got:    92
# Success. Expected:   102, Got:   102
# Success. Expected:   112, Got:   112
# Success. Expected:   122, Got:   122
# Success. Expected:   132, Got:   132
# Success. Expected:   142, Got:   142
# Success. Expected:   152, Got:   152
# Success. Expected:   162, Got:   162
```

Flow Summary	
<<Filter>>	
Flow Status	Successful - Mon May 18 16:35:30 2020
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	part1
Top-level Entity Name	part2
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	497 / 32,070 (2 %)
Total registers	621
Total pins	14 / 457 (3 %)
Total virtual pins	0
Total block memory bits	12,288 / 4,065,280 (< 1 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	1 / 6 (17 %)
Total DLLs	0 / 4 (0 %)

Task #3

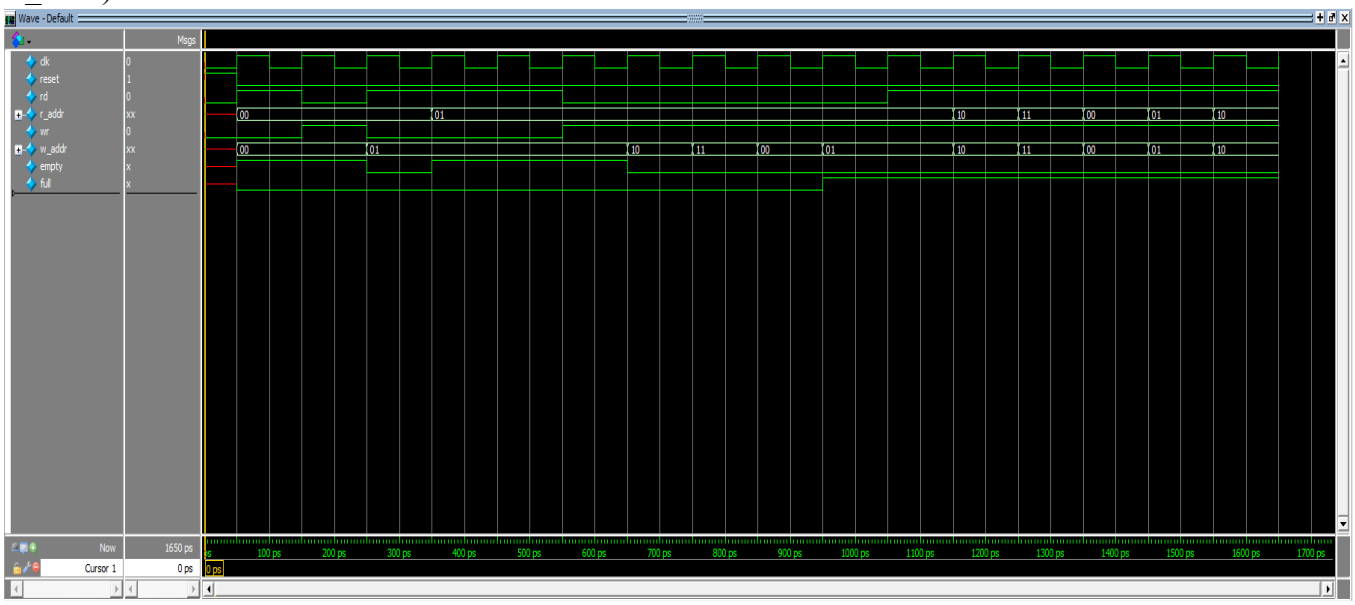
First, the *reg_file* module from *FIFO* is tested to verify correctness. The testbench simply writes 0xAA, 0xBB, 0xCC, and 0xDD in addresses 0, 1, 2, 3. Then, these addresses are read to verify their contents:



The module *FIFO_ctrl* is tested next. The testbench follows the outlined procedure:

1. Tries to read when empty, so the read address stays still (150ps).
2. Writes, and *w_addr* increases (250ps).
3. Reads an item, so *r_addr* increases (350ps).
4. Since now the FIFO should be empty, the next two reads don't increase the *r_addr*.
5. The buffer gets filled (650ps to 950p) through consecutive writes, and the full flag is asserted. The *w_addr* signal can't increase for the write at 1050ps.

6. Then both read and write are performed at the same time for a few times and both pointers (*r_addr, w_addr*) increase.

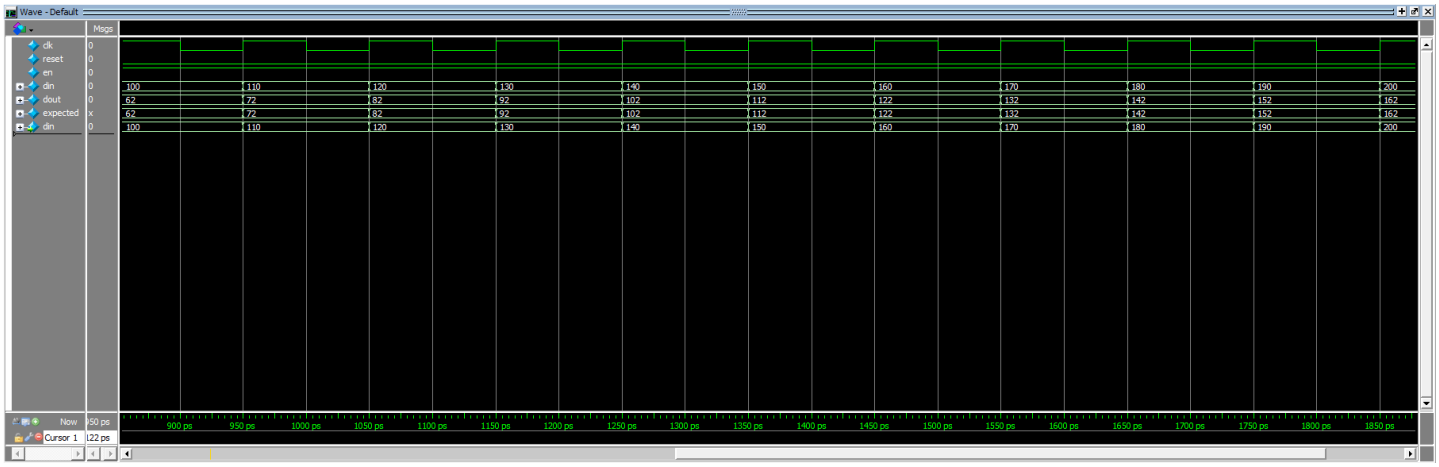


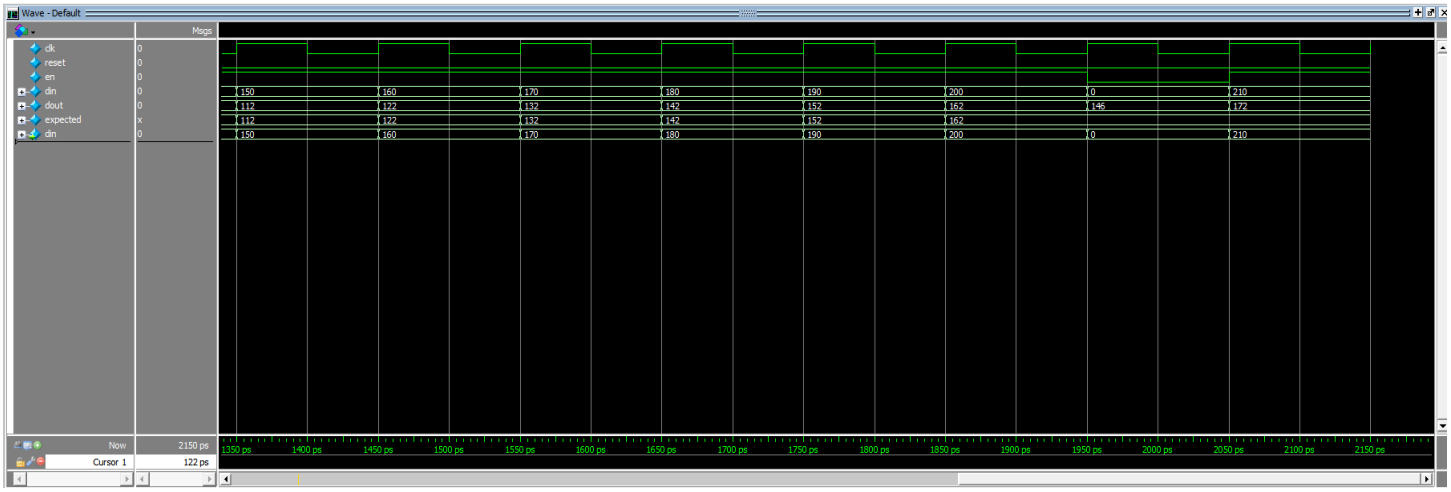
The whole *FIFO* module is tested by the following operation:

1. Writes 0 through 9.
2. Reads consecutively, which effectively returns the values 0 through 9 in a first-in first-out fashion.

Finally, having confidence in our internally-used modules we can test *FIR_Fifo* taking advantage of the same testing routine used in task 2 by selecting the parametrized width to be 8:

1. First starting values are loaded by pushing 30, 40, 50, 60, 70, 80, and 90 into the registers (omitted since the testing occurs after 850ps).
2. When 100 is pushed as input data, the average becomes 62. The average is 62 because we have 30/8+40/8+50/8+60/8+70/8+80/8+90/8+100/8=3+5+6+7+8+10+11+12=62. The real average of these numbers is 65, but the spec shows division performed before the additions which introduces a small truncation error (notice however that the bigger the numbers, the smaller the percent error thus these 24-bit numbers should not be of concern).
3. Then 110 is shifted in, the average becomes 72 as expected.
4. Similarly, the process is continued by pushing every time a number 10 units larger, until we arrive at 200 with an average of 162.
5. At the end, *en* becomes FALSE for one clock cycle and *din* is loaded with 0. Thus the output data is invalid for that clock cycle. Then, when the data becomes valid and *en* is TRUE, the new average is computed and the data that was invalid isn't taken into account.






```
# Success. Expected:     62, Got:     62
# Success. Expected:     72, Got:     72
# Success. Expected:     82, Got:     82
# Success. Expected:     92, Got:     92
# Success. Expected:    102, Got:    102
# Success. Expected:    112, Got:    112
# Success. Expected:    122, Got:    122
# Success. Expected:    132, Got:    132
# Success. Expected:    142, Got:    142
# Success. Expected:    152, Got:    152
# Success. Expected:    162, Got:    162
```

Flow Summary

<<Filter>>

Flow Status	Successful - Mon May 18 16:41:31 2020
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	part1
Top-level Entity Name	part3
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	903 / 32,070 (3 %)
Total registers	1617
Total pins	14 / 457 (3 %)
Total virtual pins	0
Total block memory bits	12,288 / 4,065,280 (< 1 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	1 / 6 (17 %)
Total DLLs	0 / 4 (0 %)

Experience Report

One challenge from this laboratory was to debug without being an expert on other parts of the design (such as the drivers for communicating audio). So far, we had learned how to get our systems working from the ground up and we had full knowledge of everything that was happening which made things easy to debug: Find the first section where an output is incorrect, and trace the source. Now, we were given a big portion of the design and we had to fit our own circuit. In order to solve this one must understand that as long as we have the specs, we can design our system to those specs. If the rest of the system works correctly and our design meets the spec, then the system will work. Thus it is evident that we just need to isolate our filters and test them with our typical testbenches until we can be sure that it meets all the specified criteria. Understanding this, we can then apply the regular debugging policy: Use your testbench to trace the source of error, fix it, and re-iterate.

Another challenge was to get the sound working for task 1. Initially, I had a microphone that I had tested on my computer (successfully). However, it would not produce sound when connected to my FPGA. After getting help during office hours, the TA suggested that getting a 3.5mm male-to-male cable could be an alternative. I was lucky to get my hands on one of those cables by borrowing from my older brother and I got my sound to work by connecting the microphone line to my phone through a 3.5mm male-to-male cable.

Regarding tips for future laboratories, what saved me from the microphone challenge was that I started early and was able to find an alternative on time. I would suggest starting as early as possible to avoid being late due to technical difficulties. Another tip is to approach the course staff as soon as you get a technical difficulty since they've had more experience and can come up with alternatives or suggestions.

Feedback about the laboratory was positive since I think it was a valuable experience to play with the audio peripherals of our DE1_SoC board. I also think the laboratory document was pretty well explained, which made each task easy to complete in a fair amount of time.

Estimated time to complete the laboratory: Around 15 hours