# Code generation of a low-pass FIR filter co-processor in FPGA

Alessandro Casalino - Badge number: 2015122
Michele Maria Crudele - Badge number: 2021858

**Abstract**

We design a low-pass FIR filter that accepts, excluding hardware limits, an integer power of 2 number of coefficients. The VHDL code to be loaded into FPGA, the testbenches code and the simulations are generated by a Python function which accepts as argument a list of coefficients. The results obtained from the testebench in VHDL, from the simulation in python and from the output of the FPGA are all consistent between each other, both for a FIR filter with 4 and 8 coefficients.

## Contents

# 1 Design

We first generate a list of random integer numbers in a text file, these will be used as input for the FPGA. On the computer, a Python code is executed to manage communication between the machine and the FPGA: it reads the random generated values from a text file and it passes them to the FPGA by a serial port, then it reads the output values generated by the FPGA and save them to an external text file.
Inside the FPGA, the input values are read by a **UART Receiver**, which passes the data to the FIR filter. The output of the FIR filter is read by a **UART Transmitter**, which sends the data back to the serial port again.
Figure 1 shows a block diagram of the described code.

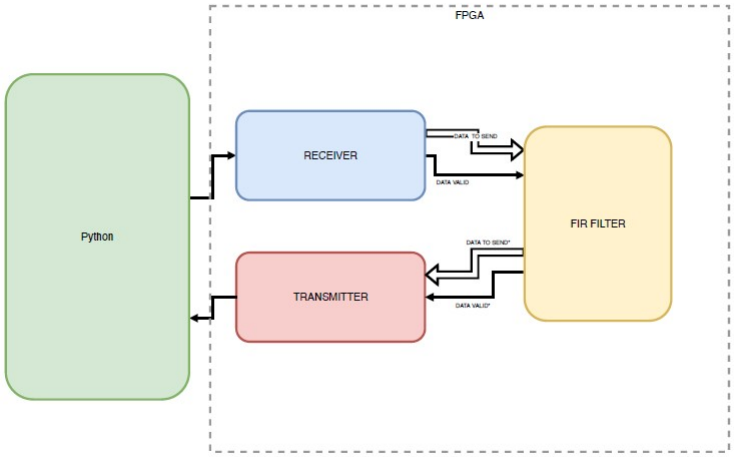



Figure 1: Block Diagram of FIR Filter implementation

# 2 Implementation

The goal of this project is to design a top entity that is composed by a Transmitter, a Receiver and a FIR Filter. The first two entities were already designed during lectures, so the first challenge is to design a FIR filter.
A Finite Impulse Response (FIR) filter takes in input n samples x[i] of a certain

length and returns in output n values y[i]. It is composed by some 'taps' C[i] and performs the following operations on the input samples:

$$y[n+1] = \sum_{i=0}^{N} x[n-i] \cdot C_i$$

where $N+1$ is the total number of taps.
The behavior of a FIR filter is determined by the number and the values of its taps, that can be decided by means of the Python function *firwin*, from the *scipy* module, which is set to compute the coefficients of a N taps digital low-pass filter with 0.1 cut-off frequency.
We started by coding a FIR filter with 4 coefficients; then, after testing its correct behaviour, we generalized the project to a FIR filter with $2^N$ coefficients and tested the correct behaviour of a FIR filter with 8 coefficients. This is done with a generator written in Python.
Taps values for the FIR filter with 4 coefficients are:
$C_0 = C_3 = 0.04566357$
$C_1 = C_2 = 0.45433643$
Taps values for the FIR filter with 8 coefficients are:
$C_0 = C_7 = 1.11400478$ $C_1 = C_6 = 3.9172728$
$C_2 = C_5 = 10.63452197$ $C_3 = C_4 = 16.33420046$
There are two main points in the construction of the FIR filter: the treatment of arithmetical operations and the handling of negative values.
Let's explain how we deal with arithmetical operations: first of all we have to convert *std_logic_vector* signals to *signed* ones, this because we can't do operation with *std_logic_vector*. In this representation, the multiplication of two arrays has length equals to the sum of the lengths of the multiplied numbers, while the result of the sum of two arrays has to be stored in an array of length that is 1 bit longer than the summed ones, in order to deal with overflow. We use the *resize* function to do all of the previous described operations.
Moreover, since the FPGA accept only 8-bits data, we have to find a way to represent the coefficients of the filter in a proper way: first of all we multiply them by two to the power of six (they are all less than one), then we approximate them to the nearest integer value and finally we represent them with 8-bits long vectors. The multiplication by the sixth power of two implies that the outputs of the filter have to be divided by the same quantity: this operation is done exploiting the *shift_right* function.
Let's now talk about how we can treat negative values into the FPGA: the input values can be in a range from -128 to 127 for representation reasons. In the Python script that communicates with the FPGA, we compute the 2's complement: we read the input value and, if it's negative, we add 256 to it, obtaining a value between 128 and 255. In fact, to pass the input values to the FPGA, we have to use *chr*() conversion in Python, and it only accepts non negative argument.
Thus the FPGA obtains values that in Python would be between 0 and 255 included. They're passed to the FIR filter as 8 bits *std_logic_vector*, but after

that they're converted into *signed*, so the most significant bit serves as sign. This way, from the point of view of the FIR filter, the input values are not in the range from 0 to 255, but from -128 to 127.
About the output of the FPGA, the negative 8-bits values are interpreted in Python as in the range from 128 to 255, so we have to reconvert them again using 2's complement.
FPGA's clock frequency is set to 100MHz. With a baudrate value of 115200 bits/s, this means the divider, i.e. the number of clock cycles with given clock frequency and baudrate, is 868.056.
In order to test the results of the FIR filter, we designed it in another environment too, different from VHDL: we choose Python 3 and we confirmed the results obtained from VHDL code.
Once designed the FIR filter, we build a **top entity**: it takes in input a sequence of bits, interpreted in group of 10 bits as the start bit, the 8-bits input values and the stop bit; these data are processed by the receiver that sends to the FIR filter input values as *std_logic_vector* plus a **data_valid** impulse that says to the filter to start working on received data.
The filter itself sends in input to the transmitter the sequence of filtered vectors, which are processed by the transmitter that gives in output the start bit, the 8 bits associated to each vector and the stop bit.
Obviously, we test each of these entities by means of a testbench.
The last goal of the project is to implement the FIR Filter co-processor in FPGA. This operation is quite simple and needs a few commands.

# 3 The VHDL code generator

We use Python language to code the generator. While code relating to receiver and transmitter is independent from the number of coefficients of the FIR filter, we need to pay attention to FIR filter's entities, top entities and related testbenches.
For the FIR filter generation, we use the *firwin* function from **scipy** library: that's the same of the FIR filter we coded entirely in Python for comparison with FPGA results. After rescaling and conversion to integer, coefficients values are used for generating VHDL in a game of for loops and formatted multi-line strings.
A particular attention has to be payed to processes that regard partial sums: for the i-th partial sum (i = 1, 2..., $log_2 n$) we need a loop from 0 to $2^{(log_2 n - i)} - 1$ over partial sum's bits, where $n$ is the number of coefficients. Furthermore, we must remember the rules on number of representation bits in multiplication and addition of values in base 2: let suppose we have two values in base 2 with same number of bits N (leading zeros are included): after adding them up we need to store the result in a (N+1) bits value, while after multiplying them with each other we need to store the result of the operation in a (2N) bits value.
The generation of top entity's code is straightforward: we just need to know the number of coefficients, from which the VHDL code is almost independent: we

used only one multi-line formatted string to change the name of top entity and FIR filter component, according to the number of coefficients chosen.
Regarding testbenches, once been able to coded for one coefficient for reference, the generalization is pretty simple and the method is similar to that for top entity generation.
VHDL code generation is modulated by a main Python script, in which we loop over a list of coefficients values and, for each value, we generate all the required VHDL code, we compile with **ghdl** the testbenches for the FIR filter and for the top entity and we create the corresponding *GTKWave* files to visualize the results.

# 4 Results and discussion

The results obtained from the testbench of the top entity in VHDL are confirmed both by the simulation in Python and by the output sent back by the FPGA, so we're sure we have designed everything in the right way.
We have come to the conclusion that the results from FPGA and Python are the same, but we need to "align" them, this because FPGA gives few more initial output values than Python implementation. From a quantitative point of view, let *cycle* be the time interval between one input value passed to the FIR filter component and the next: each output value of the FIR filter component is delayed by 4 cycles from the corresponding input value if we use 4 coefficients, by 5 cycles if we use 8 coefficients, and so on in a linear trend. Therefore, if $n$ is the number of coefficients, this delay is of $(2 + \log_2 n)$ cycles.
About the behaviour with different number of coefficients, the FIR filter seems to act more likely as an ideal low-pass filter if we increase the number of taps. We show some results in next subsection.

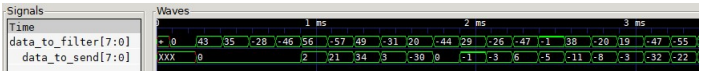
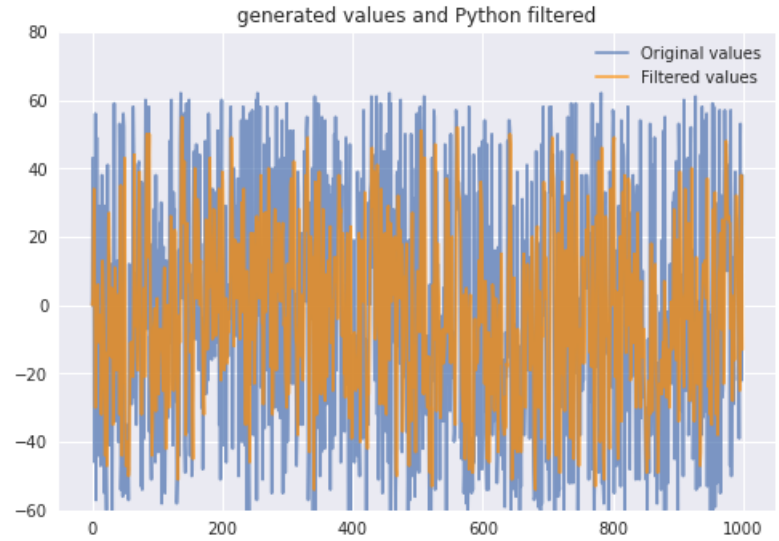


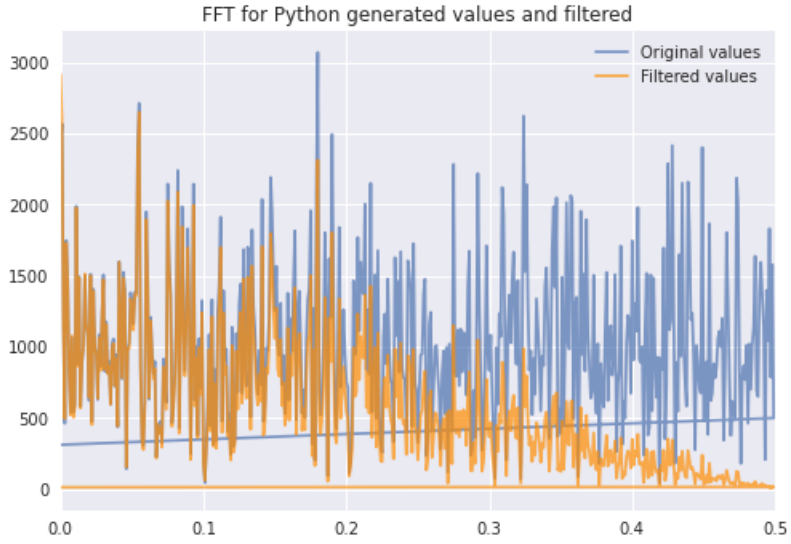

Figure 2: Input and output values from the testbench simulation for the 4 coefficients FIR filter. To avoid misinterpretations, we remind that the first "true" output values of this FIR filter are (0, 2, 21, 34,...)

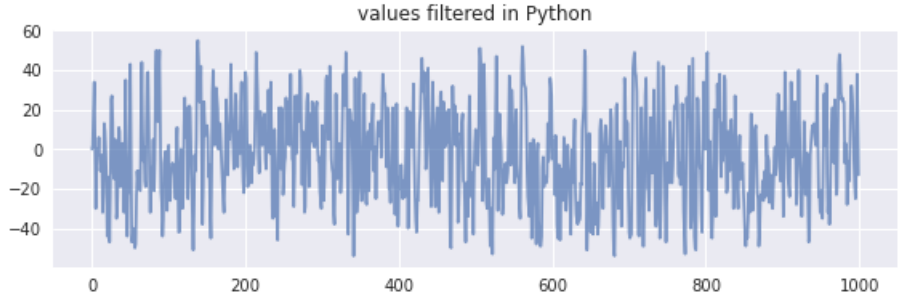
## 4.1 Comparison between FPGA results and Python results

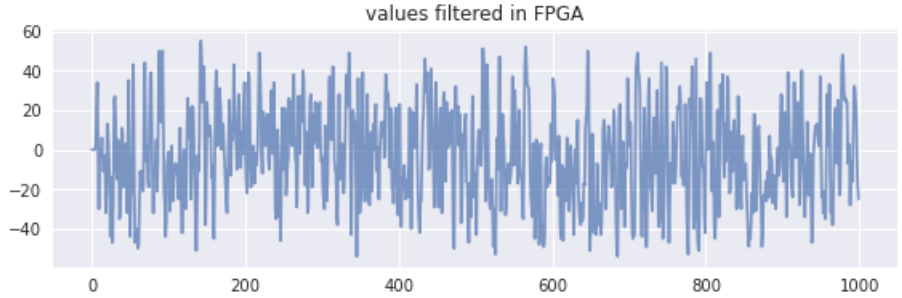
While implementing the FIR filters for the FPGA, we implemented the same FIR filters in Python for comparison purpose. Comparing the two different implementations, we could see that they're are identical, so we can be sure that the FIR filter implemented for the FPGA is reliable.

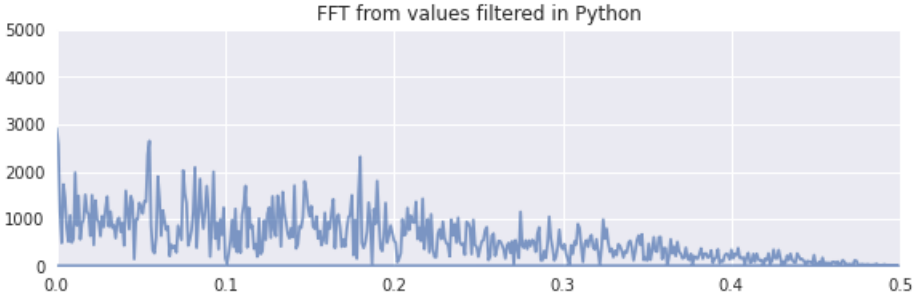
### 4.1.1 FIR filter with 4 coefficients

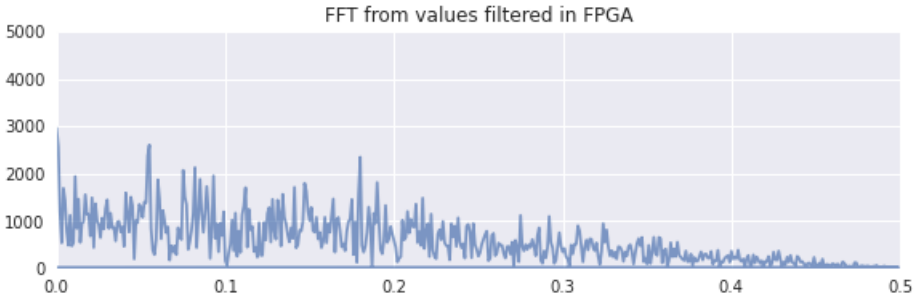


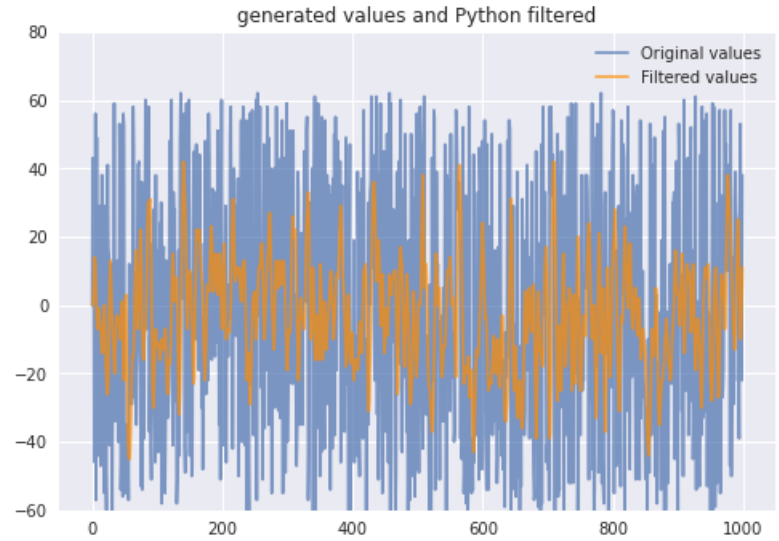

values filtered in Python
60
40
20
0
−20
−40
0
200
400
600
800
1000

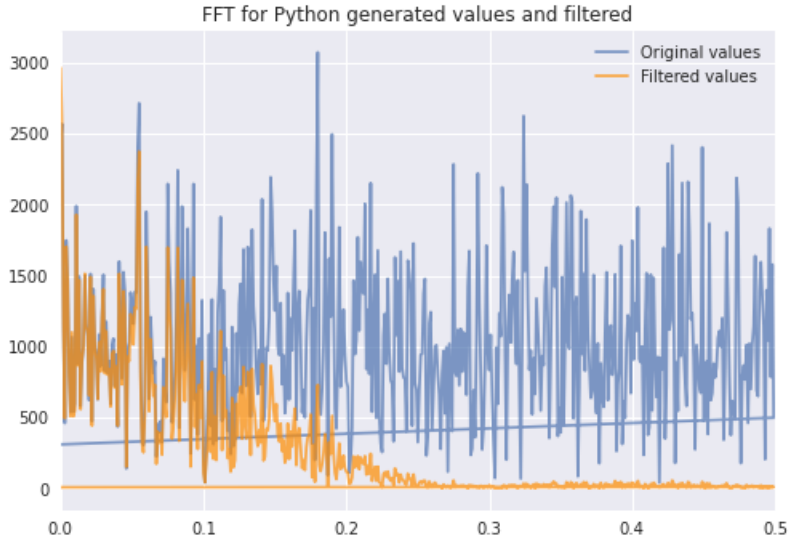

values filtered in FPGA
60
40
20
0
−20
−40
0
200
400
600
800
1000

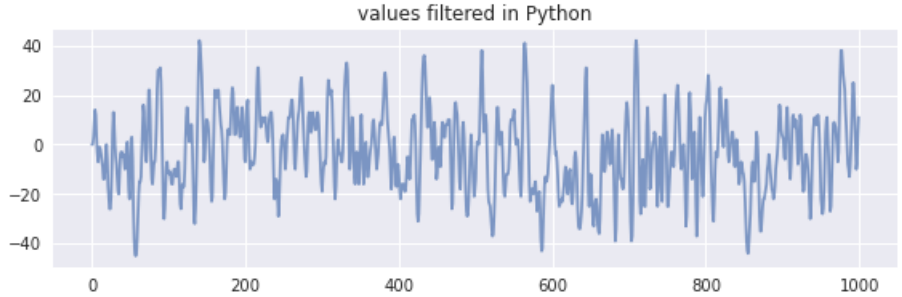

FFT from values filtered in Python
5000
4000
3000
2000
1000
0
0.0
0.1
0.2
0.3
0.4
0.5

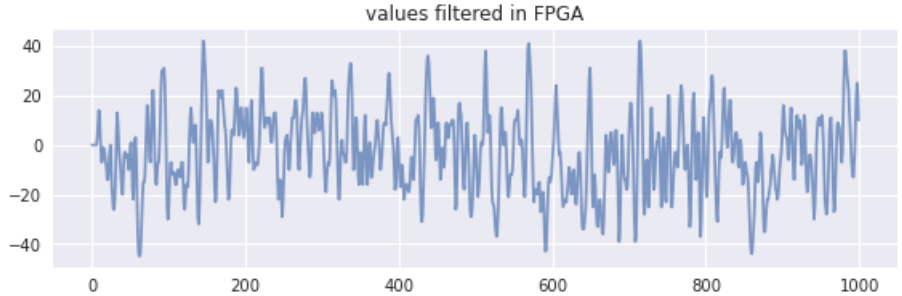

FFT from values filtered in FPGA
5000
4000
3000
2000
1000
0
0.0
0.1
0.2
0.3
0.4
0.5

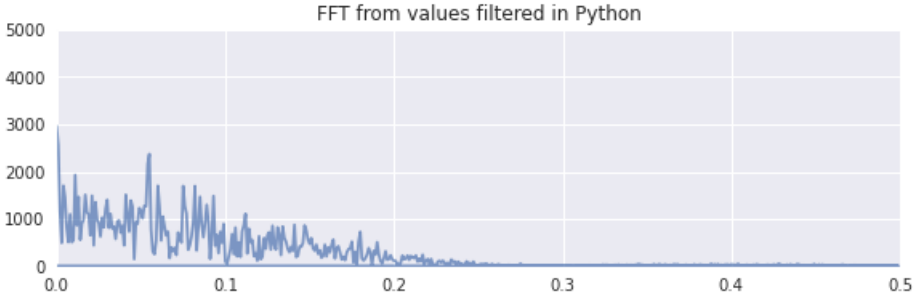
### 4.1.2 FIR filter with 8 coefficients

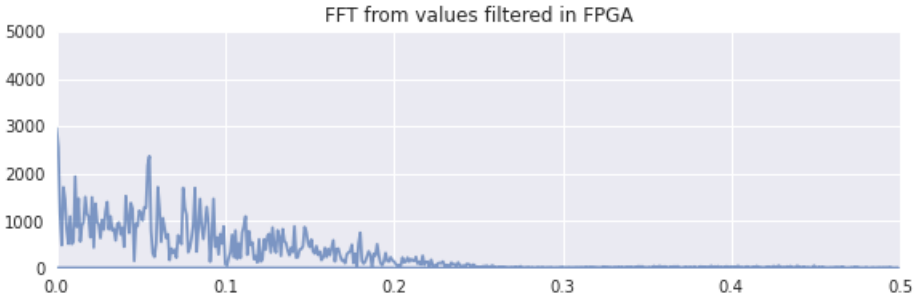



values filtered in Python
40
20
0
−20
−40
0
200
400
600
800
1000


values filtered in FPGA
40
20
0
−20
−40
0
200
400
600
800
1000


FFT from values filtered in Python
5000
4000
3000
2000
1000
0
0.0
0.1
0.2
0.3
0.4
0.5


FFT from values filtered in FPGA
5000
4000
3000
2000
1000
0
0.0
0.1
0.2
0.3
0.4
0.5

# 5 Further implementation

There are several improvements still to be done. One of these is getting better output precision sending more than one 8bits packet at time for each output value, this obviously makes FPGA implementation considerably slower respect to current version, but it could be still negligible.
The number of taps doesn't strictly have to be a integer power of 2, but we need to pay further attention in FIR filter entity implementation. We show an example to explain a way of doing that: let's suppose we want to use 7 coefficients, we can construct the partial sums in a way that we subdivide the group of coefficients into a group of 4 taps and another of 3 taps. Then, the group of 4 taps will continue as usual, while the other has to be subdivided with the same attention given to the initial group. In this way, we will eventually end with a spare coefficient, that we can appropriately sum with the others.

# A Comparison between FPGA output and Python implementation's output

## A.1 FPGA output for the 4 coefficients FIR filter

[ 0. 0. 0. 0. 0. 2. 21. 34. 3. -30. 0. -1. -3. 6. -5. -11. -8. -3. -32. -22. 13. 9.
-1. -17. -44. -17. -14. -47. -19. 25. 27. 14. -12. -15. 5. -5. -35. -30. -15. 11. 9.
-19. -6. 1. 3. -15. -32. 7. 35. -9. -44. -23. -22. 2. 43. -1. -47. -45. -40. -45.
-50. -48. -24. -11. -16. -11. -20. -21. 17. 44. 11. -6. 1. -16. -19. 14. 38. 39.
15. -15. -32. -12. -10. -8. 5. -21. -5. 38. 50. 23. 14. 40. 50. 11. -21. -27. -44.
-36. -7. -4. -1. -4. -31. -7. 2. -18. -26. -15. -10. -7. -9. -8. -15. -25. 3. 11.
-11. -42. -42. -13. 0. -19. -30. -22. 11. 26. 6. 21. 14. -25. -4. 22. 21. 0. -18.
-34. -51. -31. -3. -11. 5. 49. 55. 45. 24. 24. 42. 6. -38. -20. 24. 12. 7. 8. 12.
6. -14. -8. -32. -45. -12. 29. 40. 24. 2. 17. 31. 26. 20. 0. 11. 13. 2. -7. -17.
-26. -32. -16. 2. 25. 10. -13. 6. 5. 18. 43. 22. 9. 2. -10. 8. 28. 24. 7. -9. 7. 9.
-2. 23. 34. -1. -16. -22. -7. 39. 36. 7. -19. -17. -7. 2. -9. -17. -2. -8. -5. 2. 23.
49. 39. 26. -1. -12. 19. 19. 7. 2. 9. 16. 18. -5. -10. 4. 6. 30. 8. -2. 34. 8. -30.
-35. -27. 10. -5. -34. -32. -46. -5. 21. -8. -10. 21. 15. -23. -15. -7. -4. 26. 4.
5. 24. 2. 14. 38. 17. -5. -6. -29. -12. 27. 7. -2. 27. 14. 21. 40. 36. 7. -18. 9.
6. -32. -29. 7. 24. 28. 4. -19. 14. 18. 4. 24. 7. 2. 1. 6. 21. 24. 1. -6. -4. -21.
-26. -30. 12. 6. -26. -17. 6. 35. 37. 25. 5. 13. 42. 27. -5. -10. -12. -33. -38.
-1. 28. 0. -15. -6. 3. 0. -22. -13. 10. 28. 38. 22. 27. 49. 37. -13. -43. -8. 20.
18. 10. 14. -14. -54. -14. 36. 0. -32. -25. -20. 20. 39. 9. -26. -26. -21. 6. 28.
-8. -27. -28. 3. 9. -21. 2. 13. 0. 15. 23. -11. -10. 1. -25. -4. 8. 14. 7. 9. 14.
-4. 22. 38. 31. 27. 28. 0. -9. 21. 3. -20. -20. -27. -18. 18. 21. -8. -19. -33.
-19. 23. -7. -39. -22. -14. -20. -8. -15. -25. -16. -17. -24. -2. 21. -12. -40. -16.
19. 14. 13. 1. 11. 33. -3. -23. -36. -42. -26. -26. -4. 13. 18. 34. 46. 36. 29.
39. 18. -10. -6. 19. 41. 16. 3. -9. -29. 1. 34. 18. -12. -29. -29. 12. 21. -13.
-32. -11. 27. 29. -10. -16. 24. 3. 1. 19. 3. 7. 15. 20. -11. -48. -50. -6. 32. 15.

15. 22. 2. 18. 5. -37. -28. -11. 8. 7. 14. 18. -16. -17. -29. -47. -17. -21. -34. 10. 27. 3. -24. -12. -12. -43. -11. 3. 10. 3. -7. -8. 7. 51. 51. 42. 10. -26. 1. 43. 20. -17. -27. -30. -20. -15. -30. -49. -31. -30. -53. -13. 6. -10. 22. 47. 2. -31. 4. 18. 8. 2. -14. -33. -27. -11. -9. -7. -17. -15. -11. 10. 37. -4. -8. 30. 19. 9. 0. -17. 8. 20. -8. -14. -15. -29. -35. -2. 41. 52. 45. 34. 32. 31. 21. -9. -24. -29. -34. 3. -6. -45. -27. 5. -12. -23. -29. -48. -2. 3. -36. -43. -49. -49. -44. -9. -4. -19. -14. -14. -17. -14. 13. -6. 0. 36. 33. 27. -5. 0. 4. -23. 7. 4. -38. -45. -8. -8. -46. -19. -10. -26. -22. 6. 2. -30. -31. -18. 2. -6. -14. -26. -43. -16. 0. -17. 4. 15. -21. -33. -36. -38. -27. -36. -33. -17. -18. 10. 16. 22. 50. 47. -5. -51. -37. -13. 8. -9. -42. -32. -33. -43. -7. -9. -38. -27. -36. -43. -39. -25. 10. 15. 8. -11. -30. -8. 3. 12. 10. -1. 1. -19. -17. 4. 20. 15. -13. -38. -52. -54. -23. 21. 23. -3. -30. -19. 4. -15. -39. -11. -7. -9. 36. 26. 2. 14. 13. -21. -42. -45. -44. -43. 0. 41. 46. 49. 39. 37. 29. -6. -43. -44. -25. -5. 21. -5. -49. -49. -13. 27. 36. 12. -20. -21. -31. -14. 16. 8. -12. -6. 30. 12. -34. -39. -35. 4. 44. -1. -45. -36. -19. 8. 42. 36. 9. -27. -47. -26. -7. -17. -15. 14. 5. -17. -17. -8. 7. 22. 21. 22. 32. 23. -12. -14. -7. -18. -4. 23. 4. -31. -42. -53. -18. 26. 17. 20. 42. 0. -40. -36. 6. 46. 4. -44. -50. -51. -15. 26. 25. 4. -9. -9. -33. -42. 1. 34. 25. 0. 12. 47. 49. 2. -21. 4. -12. -40. -45. -33. 1. 20. -5. -34. 0. 33. 17. 25. 38. -5. -14. 13. -6. 0. 37. 35. 7. -22. -20. 9. 2. -11. 12. 15. 3. -7. -30. -8. 27. 7. -17. -21. -30. -2. 7. -15. -15. -9. -7. -22. -31. -43. -49. -42. -45. -36. -22. -32. 0. 18. -18. -31. -20. 8. 10. 12. -2. -22. -25. -49. -42. -24. -23. -24. -16. -26. -18. -11. -24. 7. -4. -4. -6. -27. -14. -13. -26. -30. -17. -6. 1. -13. -9. 14. 28. 27. -4. -16. 17. 19. -9. -34. -3. 39. 20. 4. 1. -15. -19. -24. 5. 34. 16. -4. -1. 16. 24. 1. -28. -22. 9. 40. 13. -6. -16. -34. -20. -12. 14. 1. -23. -4. -2. -23. -47. -43. -3. 3. 9. 12. 11. 13. 6. 2. 6. 37. 2. -24. -22. -32. -21. -33. -35. -2. 28. 2. 3. 33. -12. -32. -24. -38. -26. 9. 12. 21. 2. -7. 25. 2. -23. -20. 9. 43. 48. 43. 29. 25. 26. 24. 24. -7. 3. 0. -28. -6. -9. -16. 9. 32. 32. 25. 16. -6. -19. -25.]

## A.2 Python output for the 4 coefficients FIR filter

[0, 2, 21, 34, 3, -30, 0, -1, -3, 6, -5, -11, -8, -3, -32, -22, 13, 9, -1, -17, -44, -17, -14, -47, -19, 25, 27, 14, -12, -15, 5, -5, -35, -30, -15, 11, 9, -19, -6, 1, 3, -15, -32, 7, 35, -9, -44, -23, -22, 2, 43, -1, -47, -45, -40, -45, -50, -48, -24, -11, -16, -11, -20, -21, 17, 44, 11, -6, 1, -16, -19, 14, 38, 39, 15, -15, -32, -12, -10, -8, 5, -21, -5, 38, 50, 23, 14, 40, 50, 11, -21, -27, -44, -36, -7, -4, -1, -4, -31, -7, 2, -18, -26, -15, -10, -7, -9, -8, -15, -25, 3, 11, -11, -42, -42, -13, 0, -19, -30, -22, 11, 26, 6, 21, 14, -25, -4, 22, 21, 0, -18, -34, -51, -31, -3, -11, 5, 49, 55, 45, 24, 24, 42, 6, -38, -20, 24, 12, 7, 8, 12, 6, -14, -8, -32, -45, -12, 29, 40, 24, 2, 17, 31, 26, 20, 0, 11, 13, 2, -7, -17, -26, -32, -16, 2, 25, 10, -13, 6, 5, 18, 43, 22, 9, 2, -10, 8, 28, 24, 7, -9, 7, 9, -2, 23, 34, -1, -16, -22, -7, 39, 36, 7, -19, -17, -7, 2, -9, -17, -2, -8, -5, 2, 23, 49, 39, 26, -1, -12, 19, 19, 7, 2, 9, 16, 18, -5, -10, 4, 6, 30, 8, -2, 34, 8, -30, -35, -27, 10, -5, -34, -32, -46, -5, 21, -8, -10, 21, 15, -23, -15, -7, -4, 26, 4, 5, 24, 2, 14, 38, 17, -5, -6, -29, -12, 27, 7, -2, 27, 14, 21, 40, 36, 7, -18, 9, 6, -32, -29, 7, 24, 28, 4, -19, 14, 18, 4, 24, 7, 2, 1, 6, 21, 24, 1, -6, -4, -21, -26, -30, 12, 6, -26, -17, 6, 35, 37, 25, 5, 13, 42, 27, -5, -10, -12, -33, -38, -1, 28, 0, -15, -6, 3,

0, -22, -13, 10, 28, 38, 22, 27, 49, 37, -13, -43, -8, 20, 18, 10, 14, -14, -54, -14, 36, 0, -32, -25, -20, 20, 39, 9, -26, -26, -21, 6, 28, -8, -27, -28, 3, 9, -21, 2, 13, 0, 15, 23, -11, -10, 1, -25, -4, 8, 14, 7, 9, 14, -4, 22, 38, 31, 27, 28, 0, -9, 21, 3, -20, -20, -27, -18, 18, 21, -8, -19, -33, -19, 23, -7, -39, -22, -14, -20, -8, -15, -25, -16, -17, -24, -2, 21, -12, -40, -16, 19, 14, 13, 1, 11, 33, -3, -23, -36, -42, -26, -26, -4, 13, 18, 34, 46, 36, 29, 39, 18, -10, -6, 19, 41, 16, 3, -9, -29, 1, 34, 18, -12, -29, -29, 12, 21, -13, -32, -11, 27, 29, -10, -16, 24, 3, 1, 19, 3, 7, 15, 20, -11, -48, -50, -6, 32, 15, 15, 22, 2, 18, 5, -37, -28, -11, 8, 7, 14, 18, -16, -17, -29, -47, -17, -21, -34, 10, 27, 3, -24, -12, -12, -43, -11, 3, 10, 3, -7, -8, 7, 51, 51, 42, 10, -26, 1, 43, 20, -17, -27, -30, -20, -15, -30, -49, -31, -30, -53, -13, 6, -10, 22, 47, 2, -31, 4, 18, 8, 2, -14, -33, -27, -11, -9, -7, -17, -15, -11, 10, 37, -4, -8, 30, 19, 9, 0, -17, 8, 20, -8, -14, -15, -29, -35, -2, 41, 52, 45, 34, 32, 31, 21, -9, -24, -29, -34, 3, -6, -45, -27, 5, -12, -23, -29, -48, -2, 3, -36, -43, -49, -49, -44, -9, -4, -19, -14, -14, -17, -14, 13, -6, 0, 36, 33, 27, -5, 0, 4, -23, 7, 4, -38, -45, -8, -8, -46, -19, -10, -26, -22, 6, 2, -30, -31, -18, 2, -6, -14, -26, -43, -16, 0, -17, 4, 15, -21, -33, -36, -38, -27, -36, -33, -17, -18, 10, 16, 22, 50, 47, -5, -51, -37, -13, 8, -9, -42, -32, -33, -43, -7, -9, -38, -27, -36, -43, -39, -25, 10, 15, 8, -11, -30, -8, 3, 12, 10, -1, 1, -19, -17, 4, 20, 15, -13, -38, -52, -54, -23, 21, 23, -3, -30, -19, 4, -15, -39, -11, -7, -9, 36, 26, 2, 14, 13, -21, -42, -45, -44, -43, 0, 41, 46, 49, 39, 37, 29, -6, -43, -44, -25, -5, 21, -5, -49, -49, -13, 27, 36, 12, -20, -21, -31, -14, 16, 8, -12, -6, 30, 12, -34, -39, -35, 4, 44, -1, -45, -36, -19, 8, 42, 36, 9, -27, -47, -26, -7, -17, -15, 14, 5, -17, -17, -8, 7, 22, 21, 22, 32, 23, -12, -14, -7, -18, -4, 23, 4, -31, -42, -53, -18, 26, 17, 20, 42, 0, -40, -36, 6, 46, 4, -44, -50, -51, -15, 26, 25, 4, -9, -9, -33, -42, 1, 34, 25, 0, 12, 47, 49, 2, -21, 4, -12, -40, -45, -33, 1, 20, -5, -34, 0, 33, 17, 25, 38, -5, -14, 13, -6, 0, 37, 35, 7, -22, -20, 9, 2, -11, 12, 15, 3, -7, -30, -8, 27, 7, -17, -21, -30, -2, 7, -15, -15, -9, -7, -22, -31, -43, -49, -42, -45, -36, -22, -32, 0, 18, -18, -31, -20, 8, 10, 12, -2, -22, -25, -49, -42, -24, -23, -24, -16, -26, -18, -11, -24, 7, -4, -4, -6, -27, -14, -13, -26, -30, -17, -6, 1, -13, -9, 14, 28, 27, -4, -16, 17, 19, -9, -34, -3, 39, 20, 4, 1, -15, -19, -24, 5, 34, 16, -4, -1, 16, 24, 1, -28, -22, 9, 40, 13, -6, -16, -34, -20, -12, 14, 1, -23, -4, -2, -23, -47, -43, -3, 3, 9, 12, 11, 13, 6, 2, 6, 37, 2, -24, -22, -32, -21, -33, -35, -2, 28, 2, 3, 33, -12, -32, -24, -38, -26, 9, 12, 21, 2, -7, 25, 2, -23, -20, 9, 43, 48, 43, 29, 25, 26, 24, 24, -7, 3, 0, -28, -6, -9, -16, 9, 32, 32, 25, 16, -6, -19, -25, 0, 38, 18, -13]

## A.3 FPGA output for the 8 coefficients FIR filter

[ 0. 0. 0. 0. 0. 0. 0. 3. 9. 14. 12. 3. -3. -7. -3. -1. -2. -4. -7. -9. -14. -14. -10. -3. 0. -6. -17. -22. -26. -26. -21. -10. 5. 13. 8. 0. -5. -8. -14. -19. -20. -12. -4. -3. -3. -5. -4. -7. -10. -7. -1. 1. -7. -18. -22. -11. 2. 3. -6. -25. -38. -44. -45. -44. -38. -29. -20. -15. -15. -13. -3. 9. 16. 13. 4. -5. -7. -1. 11. 22. 22. 11. -6. -14. -16. -11. -7. -7. -1. 8. 22. 30. 30. 30. 31. 25. 11. -9. -25. -30. -26. -17. -9. -7. -11. -11. -12. -11. -14. -16. -16. -12. -10. -10. -11. -13. -10. -7. -7. -15. -24. -26. -21. -16. -17. -17. -10. 2. 11. 15. 11. 4. 1. 2. 8. 8. -2. -16. -29. -32. -26. -15. -1. 16. 32. 42. 40. 34. 28. 18. 4. -7. -6. 2. 10. 10. 9. 6. 0. -7. -18. -23. -21. -6. 12. 22. 21. 19. 19. 22. 22. 16. 12. 8. 6. 1. -8. -16. -22. -20. -12. 0. 6. 6. 4. 4. 11. 19. 23. 21. 12. 4. 4. 9. 15. 15. 9. 5. 3. 6. 11. 15. 12. 4. -6.

-7. 4. 15. 18. 7. -4. -10. -9. -7. -8. -8. -8. -5. 0. 10. 23. 31. 31. 21. 10. 7. 8. 11. 10. 9. 10. 11. 7. 3. 1. 3. 10. 12. 13. 13. 8. -1. -14. -22. -17. -14. -15. -22. -29. -23. -12. -3. 2. 3. 4. 1. -5. -10. -6. 2. 6. 11. 11. 11. 15. 18. 18. 13. 2. -8. -10. -4. 3. 9. 12. 14. 20. 25. 27. 22. 11. 4. -3. -8. -13. -11. 0. 11. 13. 7. 5. 5. 10. 13. 11. 9. 6. 6. 10. 13. 12. 6. -2. -10. -16. -19. -13. -9. -7. -8. -5. 8. 20. 26. 23. 20. 21. 22. 17. 5. -7. -17. -22. -17. -6. 1. 2. -2. -4. -4. -7. -7. -3. 9. 21. 27. 31. 33. 30. 18. 0. -10. -7. 5. 11. 10. -1. -13. -16. -10. -3. -4. -13. -16. -6. 7. 12. 4. -9. -16. -10. -1. 1. -4. -13. -13. -9. -5. -2. 0. 4. 9. 10. 6. 1. -5. -10. -7. -4. 3. 8. 9. 9. 8. 13. 19. 27. 29. 26. 17. 10. 6. 2. -2. -11. -18. -16. -7. 2. 3. -4. -14. -17. -11. -8. -11. -18. -22. -19. -16. -16. -16. -18. -19. -17. -12. -5. -5. -10. -14. -8. 2. 10. 10. 11. 12. 8. -1. -17. -28. -31. -28. -17. -5. 8. 21. 30. 35. 36. 33. 25. 15. 7. 7. 15. 19. 15. 4. -6. -6. 2. 9. 6. -5. -14. -11. -4. -1. -6. -9. -3. 7. 9. 5. 3. 3. 8. 8. 8. 10. 10. 10. 1. -13. -26. -24. -9. 7. 16. 17. 14. 12. 4. -6. -15. -18. -10. -1. 7. 9. 3. -6. -19. -27. -29. -28. -22. -12. -2. 4. 0. -8. -15. -21. -18. -14. -3. 1. 1. 0. 4. 18. 31. 38. 29. 13. 5. 8. 12. 8. -6. -18. -23. -24. -25. -30. -34. -37. -36. -29. -20. -7. 6. 15. 15. 7. 0. 0. 5. 5. -3. -13. -20. -21. -17. -12. -12. -12. -10. -3. 7. 10. 10. 10. 12. 14. 8. 2. 0. 2. 2. -2. -11. -18. -21. -14. 4. 25. 39. 41. 38. 32. 25. 12. -3. -16. -23. -19. -18. -18. -20. -19. -16. -15. -21. -27. -23. -20. -19. -26. -38. -43. -42. -32. -22. -15. -13. -15. -14. -12. -7. -3. 4. 12. 20. 24. 17. 9. 1. -4. -3. -7. -13. -21. -25. -24. -23. -22. -23. -20. -17. -12. -9. -11. -17. -20. -16. -11. -10. -16. -23. -24. -20. -12. -5. -3. -6. -14. -26. -33. -34. -34. -31. -27. -20. -8. 3. 16. 27. 31. 21. -1. -20. -25. -17. -12. -17. -25. -32. -33. -27. -23. -22. -25. -32. -35. -36. -30. -17. -4. 4. 1. -8. -11. -9. 0. 5. 5. 2. -5. -8. -6. 1. 6. 1. -13. -30. -39. -34. -18. -2. 4. -3. -11. -14. -14. -17. -18. -16. -7. 6. 13. 17. 14. 8. -2. -17. -31. -39. -37. -23. 0. 23. 38. 42. 40. 31. 15. -6. -23. -28. -21. -9. -6. -14. -25. -25. -10. 8. 15. 7. -7. -18. -17. -9. -2. 1. 1. 5. 4. -3. -16. -25. -17. -2. 3. -5. -19. -23. -12. 7. 20. 20. 4. -15. -25. -25. -19. -12. -6. -3. -3. -7. -9. -4. 6. 15. 21. 24. 21. 12. 2. -7. -10. -7. -1. 0. -7. -21. -33. -29. -14. 4. 17. 21. 14. 0. -14. -13. 0. 5. -4. -24. -37. -31. -14. 4. 11. 6. -5. -16. -21. -16. -3. 11. 16. 18. 23. 28. 25. 12. 0. -11. -19. -28. -31. -22. -10. -3. -5. -5. 2. 13. 22. 23. 16. 8. 2. -1. 5. 12. 18. 18. 7. -4. -7. -5. -1. 3. 5. 5. 1. -8. -9. -3. 2. 0. -8. -16. -15. -11. -7. -8. -11. -12. -15. -22. -31. -39. -43. -44. -40. -34. -28. -17. -9. -7. -10. -15. -11. -3. 5. 3. -5. -17. -29. -35. -35. -30. -25. -22. -22. -20. -18. -16. -10. -8. -4. -6. -11. -14. -18. -19. -22. -21. -17. -10. -7. -4. 1. 10. 16. 13. 6. 4. 3. 2. -5. -6. 2. 11. 15. 8. -2. -10. -14. -8. 3. 11. 12. 8. 7. 10. 7. -1. -8. -6. 6. 12. 10. -2. -15. -19. -18. -9. -4. -4. -6. -10. -15. -23. -30. -25. -15. -1. 6. 10. 11. 9. 8. 8. 12. 9. 2. -11. -22. -25. -28. -26. -18. -5. 5. 10. 11. 3. -6. -19. -27. -25. -15. -3. 8. 9. 8. 7. 3. -1. -7. -3. 12. 28. 38. 38. 33. 28. 25. 22. 13. 7. -1. -8. -10. -13. -9. -2. 9. 20. 25. 21. 10.]

## A.4 Python output for the 8 coefficients FIR filter

[0, 0, 3, 9, 14, 12, 3, -3, -7, -3, -1, -2, -4, -7, -9, -14, -14, -10, -3, 0, -6, -17, -22, -26, -26, -21, -10, 5, 13, 8, 0, -5, -8, -14, -19, -20, -12, -4, -3, -3, -5, -4, -7, -10, -7, -1, 1, -7, -18, -22, -11, 2, 3, -6, -25, -38, -44, -45, -44, -38, -29, -20, -15, -15,

-13, -3, 9, 16, 13, 4, -5, -7, -1, 11, 22, 22, 11, -6, -14, -16, -11, -7, -7, -1, 8, 22, 30, 30, 30, 31, 25, 11, -9, -25, -30, -26, -17, -9, -7, -11, -11, -12, -11, -14, -16, -16, -12, -10, -10, -11, -13, -10, -7, -7, -15, -24, -26, -21, -16, -17, -17, -10, 2, 11, 15, 11, 4, 1, 2, 8, 8, -2, -16, -29, -32, -26, -15, -1, 16, 32, 42, 40, 34, 28, 18, 4, -7, -6, 2, 10, 10, 9, 6, 0, -7, -18, -23, -21, -6, 12, 22, 21, 19, 19, 22, 22, 16, 12, 8, 6, 1, -8, -16, -22, -20, -12, 0, 6, 6, 4, 4, 11, 19, 23, 21, 12, 4, 4, 9, 15, 15, 9, 5, 3, 6, 11, 15, 12, 4, -6, -7, 4, 15, 18, 7, -4, -10, -9, -7, -8, -8, -8, -5, 0, 10, 23, 31, 31, 21, 10, 7, 8, 11, 10, 9, 10, 11, 7, 3, 1, 3, 10, 12, 13, 13, 8, -1, -14, -22, -17, -14, -15, -22, -29, -23, -12, -3, 2, 3, 4, 1, -5, -10, -6, 2, 6, 11, 11, 11, 15, 18, 18, 13, 2, -8, -10, -4, 3, 9, 12, 14, 20, 25, 27, 22, 11, 4, -3, -8, -13, -11, 0, 11, 13, 7, 5, 5, 10, 13, 11, 9, 6, 6, 10, 13, 12, 6, -2, -10, -16, -19, -13, -9, -7, -8, -5, 8, 20, 26, 23, 20, 21, 22, 17, 5, -7, -17, -22, -17, -6, 1, 2, -2, -4, -4, -7, -7, -3, 9, 21, 27, 31, 33, 30, 18, 0, -10, -7, 5, 11, 10, -1, -13, -16, -10, -3, -4, -13, -16, -6, 7, 12, 4, -9, -16, -10, -1, 1, -4, -13, -13, -9, -5, -2, 0, 4, 9, 10, 6, 1, -5, -10, -7, -4, 3, 8, 9, 9, 8, 13, 19, 27, 29, 26, 17, 10, 6, 2, -2, -11, -18, -16, -7, 2, 3, -4, -14, -17, -11, -8, -11, -18, -22, -19, -16, -16, -16, -18, -19, -17, -12, -5, -5, -10, -14, -8, 2, 10, 10, 11, 12, 8, -1, -17, -28, -31, -28, -17, -5, 8, 21, 30, 35, 36, 33, 25, 15, 7, 7, 15, 19, 15, 4, -6, -6, 2, 9, 6, -5, -14, -11, -4, -1, -6, -9, -3, 7, 9, 5, 3, 3, 8, 8, 8, 10, 10, 10, 1, -13, -26, -24, -9, 7, 16, 17, 14, 12, 4, -6, -15, -18, -10, -1, 7, 9, 3, -6, -19, -27, -29, -28, -22, -12, -2, 4, 0, -8, -15, -21, -18, -14, -3, 1, 1, 0, 4, 18, 31, 38, 29, 13, 5, 8, 12, 8, -6, -18, -23, -24, -25, -30, -34, -37, -36, -29, -20, -7, 6, 15, 15, 7, 0, 0, 5, 5, -3, -13, -20, -21, -17, -12, -12, -12, -10, -3, 7, 10, 10, 10, 12, 14, 8, 2, 0, 2, 2, -2, -11, -18, -21, -14, 4, 25, 39, 41, 38, 32, 25, 12, -3, -16, -23, -19, -18, -18, -20, -19, -16, -15, -21, -27, -23, -20, -19, -26, -38, -43, -42, -32, -22, -15, -13, -15, -14, -12, -7, -3, 4, 12, 20, 24, 17, 9, 1, -4, -3, -7, -13, -21, -25, -24, -23, -22, -23, -20, -17, -12, -9, -11, -17, -20, -16, -11, -10, -16, -23, -24, -20, -12, -5, -3, -6, -14, -26, -33, -34, -34, -31, -27, -20, -8, 3, 16, 27, 31, 21, -1, -20, -25, -17, -12, -17, -25, -32, -33, -27, -23, -22, -25, -32, -35, -36, -30, -17, -4, 4, 1, -8, -11, -9, 0, 5, 5, 2, -5, -8, -6, 1, 6, 1, -13, -30, -39, -34, -18, -2, 4, -3, -11, -14, -14, -17, -18, -16, -7, 6, 13, 17, 14, 8, -2, -17, -31, -39, -37, -23, 0, 23, 38, 42, 40, 31, 15, -6, -23, -28, -21, -9, -6, -14, -25, -25, -10, 8, 15, 7, -7, -18, -17, -9, -2, 1, 1, 5, 4, -3, -16, -25, -17, -2, 3, -5, -19, -23, -12, 7, 20, 20, 4, -15, -25, -25, -19, -12, -6, -3, -3, -7, -9, -4, 6, 15, 21, 24, 21, 12, 2, -7, -10, -7, -1, 0, -7, -21, -33, -29, -14, 4, 17, 21, 14, 0, -14, -13, 0, 5, -4, -24, -37, -31, -14, 4, 11, 6, -5, -16, -21, -16, -3, 11, 16, 18, 23, 28, 25, 12, 0, -11, -19, -28, -31, -22, -10, -3, -5, -5, 2, 13, 22, 23, 16, 8, 2, -1, 5, 12, 18, 18, 7, -4, -7, -5, -1, 3, 5, 5, 1, -8, -9, -3, 2, 0, -8, -16, -15, -11, -7, -8, -11, -12, -15, -22, -31, -39, -43, -44, -40, -34, -28, -17, -9, -7, -10, -15, -11, -3, 5, 3, -5, -17, -29, -35, -35, -30, -25, -22, -22, -20, -18, -16, -10, -8, -4, -6, -11, -14, -18, -19, -22, -21, -17, -10, -7, -4, 1, 10, 16, 13, 6, 4, 3, 2, -5, -6, 2, 11, 15, 8, -2, -10, -14, -8, 3, 11, 12, 8, 7, 10, 7, -1, -8, -6, 6, 12, 10, -2, -15, -19, -18, -9, -4, -4, -6, -10, -15, -23, -30, -25, -15, -1, 6, 10, 11, 9, 8, 8, 12, 9, 2, -11, -22, -25, -28, -26, -18, -5, 5, 10, 11, 3, -6, -19, -27, -25, -15, -3, 8, 9, 8, 7, 3, -1, -7, -3, 12, 28, 38, 38, 33, 28, 25, 22, 13, 7, -1, -8, -10, -13, -9, -2, 9, 20, 25, 21, 10, -2, -10, -8, 3, 11]

# B Testbenches simulations

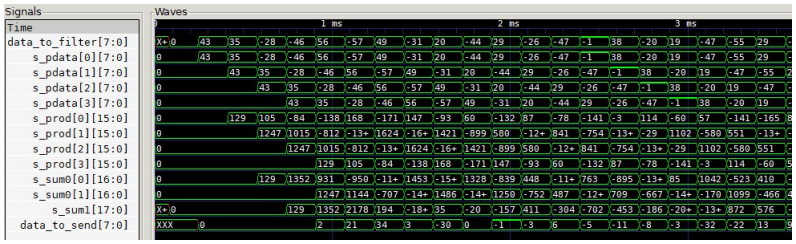



Figure 3: Simulation for the testbench for top entity of the 4 coefficients FIR filter


Figure 4: Simulation for the testbench for top entity of the 8 coefficients FIR filter

# C Codes

## C.1 FIR filter code generator

```
import math
from scipy.signal import firwin

#=======================================================
# Function for generating FIR filter .vhd files
#=======================================================
def generate_fir(n: int):
    log2_n = int(round(math.log(n)/math.log(2)))
    # We only accept n = 2^k for how algorithm works
    if 2**log2_n != n:
        print(f"Error: {n} is not a valid parameter. Exiting.")
        return
    # List of strings to be written on file
    s_list = []
    # Libraries
    s_temp = """\
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

"""
    s_list.append(s_temp)
    # Entity name
    s_list.append(f"entity fir{n} is\n")
    # Clock and Reset
    s_temp = """\
   port(
      -- clock
      clock: in std_logic;
      -- to know when we can accept input data
      data_valid: in std_logic;
      -- coefficients used for FIR filter
"""
    s_list.append(s_temp)
    # Rest of entity and first part of architecture
    s_temp = f"""\
      -- input data
      i_data: in std_logic_vector(7 downto 0);
      -- ouput (filtered) data
      o_data: out std_logic_vector(7 downto 0)
   );
end entity fir{n};

architecture rtl of fir{n} is
```

```
    -- we declare types because type mark is expected in a subtype
        indication
    -- all the coefficients
    type t_coeffs is array (0 to {n-1}) of signed(7 downto 0);
"""
    s_list.append(s_temp)
    # Coefficients' values
    s_temp = "constant s_coeffs: t_coeffs := ("
    f = 0.1
    coeffs = firwin(n, f)
    coeffs = [round(c * 2**6) for c in coeffs]
    for i in range(n):
        s_temp += f"to_signed({coeffs[i]}, 8)"
        if i < n-1:
            s_temp += ", "
        else:
            s_temp += ");\n"
    s_list.append(s_temp)
    # Remaining signals
    s_temp = f"""\
    -- all the piped data needed for computation (num piped data = num
        coefficients)
    type t_pdata is array (0 to {n-1}) of signed(7 downto 0);
    signal s_pdata: t_pdata := (others=>(to_signed(0, 8)));
    -- all the products used for the sum
    -- when multiplying two numbers of n bits, we need 2n bits to
        represent the sum
    type t_prod is array(0 to {n-1}) of signed(8*2-1 downto 0);
    signal s_prod: t_prod := (others=>(to_signed(0, 8*2)));
    -- we sum with a binary algorithm
"""
    s_list.append(s_temp)
    # Signals used for sum
    s_temp = ""
    # we avoid last s_sum in the loop because it doesn't need type
        definition
    for i in range(log2_n-1):
        s_temp += f"""\
    type t_sum{i} is array(0 to {2**(log2_n - i - 1)-1}) of
        signed(8*2+{i} downto 0);
    signal s_sum{i}: t_sum{i} := (others=>(to_signed(0, 8*2+{i}+1)));
"""
    s_list.append(s_temp)
    # Last sum signal
    s_list.append(f" signal s_sum{log2_n-1}: signed(8*2+{log2_n-1}
        downto 0);\n\n")
    # Process part (initialization)
    s_temp = """\
begin
    p_input: process(clock) is
```

```
  begin
    if rising_edge(clock) then
      if data_valid = '1' then
        -- pipe data
        s_pdata <= signed(i_data) & s_pdata(0 to s_pdata'length-2);
"""
    s_list.append(s_temp)
    # Product process
    s_temp = f"""\
      end if;
    end if;
  end process p_input;

  p_prod: process(clock) is
  begin
    if rising_edge(clock) then
      if data_valid = '1' then
        for i in 0 to {n-1} loop
          s_prod(i) <= s_pdata(i) * s_coeffs(i);
        end loop;
      end if;
    end if;
  end process p_prod;

"""
    s_list.append(s_temp)
    # First process for sum
    s_temp = f"""\
  p_sum0: process(clock) is
  begin
    if rising_edge(clock) then
      if data_valid = '1' then
        for i in 0 to {(n//2) - 1} loop
          s_sum0(i) <= resize(s_prod(2*i), 2*8+1)
            + resize(s_prod(2*i+1), 2*8+1);
        end loop;
      end if;
    end if;
  end process p_sum0;

"""
    s_list.append(s_temp)
    # Processes for other sums except last
    for i in range(1, log2_n-1):
        s_temp = f"""\
  p_sum{i}: process(clock) is
  begin
    if rising_edge(clock) then
      if data_valid = '1' then
            for i in 0 to {2**(log2_n-1 - i) - 1} loop
```

```
                s_sum{i}(i) <= resize(s_sum{i-1}(2*i), 2*8+1+{i})
                    + resize(s_sum{i-1}(2*i+1), 2*8+1+{i});
             end loop;
        end if;
      end if;
   end process p_sum{i};

"""
       s_list.append(s_temp)
    # Process for last sum
    s_temp = f"""\
   p_sum{log2_n-1}: process(clock) is
   begin
      if rising_edge(clock) then
         if data_valid = '1' then
            s_sum{log2_n-1} <= resize(s_sum{log2_n-2}(0), 2*8+{log2_n})
               + resize(s_sum{log2_n-2}(1), 2*8+{log2_n});
         end if;
      end if;
   end process p_sum{log2_n-1};

"""
    s_list.append(s_temp)
    # Process for the output and "ending" architecture
    s_temp = f"""\
   p_output: process(clock) is
   begin
      if rising_edge(clock) then
         if data_valid = '1' then
            -- we must lose precision in the 8bits output
            o_data <= std_logic_vector( shift_right(s_sum{log2_n-1}, 6)(7
                downto 0) );
         end if;
      end if;
   end process p_output;
end architecture rtl;
"""
    s_list.append(s_temp)
    #======================================================
    # Writing on file
    #======================================================
    with open(f"generated_code/fir{n}.vhd", "w") as fid:
        for s in s_list:
            fid.write(s)
```

## C.2 FIR Filter with 8 coefficients

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity fir8 is
  port(
    -- clock
    clock: in std_logic;
    -- to know when we can accept input data
    data_valid: in std_logic;
    -- coefficients used for FIR filter
    -- input data
    i_data: in std_logic_vector(7 downto 0);
    -- ouput (filtered) data
    o_data: out std_logic_vector(7 downto 0)
  );
end entity fir8;

architecture rtl of fir8 is
  -- we declare types because type mark is expected in a subtype
      indication
  -- all the coefficients
  type t_coeffs is array (0 to 7) of signed(7 downto 0);
constant s_coeffs: t_coeffs := (to_signed(1, 8), to_signed(4, 8),
    to_signed(11, 8), to_signed(16, 8), to_signed(16, 8), to_signed(11,
    8), to_signed(4, 8), to_signed(1, 8));
  -- all the piped data needed for computation (num piped data = num
      coefficients)
  type t_pdata is array (0 to 7) of signed(7 downto 0);
  signal s_pdata: t_pdata := (others=>(to_signed(0, 8)));
  -- all the products used for the sum
  -- when multiplying two numbers of n bits, we need 2n bits to
      represent the sum
  type t_prod is array(0 to 7) of signed(8*2-1 downto 0);
  signal s_prod: t_prod := (others=>(to_signed(0, 8*2)));
  -- we sum with a binary algorithm
   type t_sum0 is array(0 to 3) of signed(8*2+0 downto 0);
   signal s_sum0: t_sum0 := (others=>(to_signed(0, 8*2+0+1)));
   type t_sum1 is array(0 to 1) of signed(8*2+1 downto 0);
   signal s_sum1: t_sum1 := (others=>(to_signed(0, 8*2+1+1)));
  signal s_sum2: signed(8*2+2 downto 0);

begin
  p_input: process(clock) is
  begin
    if rising_edge(clock) then
      if data_valid = '1' then
```

```
        -- pipe data
        s_pdata <= signed(i_data) & s_pdata(0 to s_pdata'length-2);
      end if;
   end if;
end process p_input;

p_prod: process(clock) is
begin
   if rising_edge(clock) then
      if data_valid = '1' then
         for i in 0 to 7 loop
            s_prod(i) <= s_pdata(i) * s_coeffs(i);
         end loop;
      end if;
   end if;
end process p_prod;

p_sum0: process(clock) is
begin
   if rising_edge(clock) then
      if data_valid = '1' then
         for i in 0 to 3 loop
            s_sum0(i) <= resize(s_prod(2*i), 2*8+1)
              + resize(s_prod(2*i+1), 2*8+1);
         end loop;
      end if;
   end if;
end process p_sum0;

p_sum1: process(clock) is
begin
   if rising_edge(clock) then
      if data_valid = '1' then
            for i in 0 to 1 loop
                s_sum1(i) <= resize(s_sum0(2*i), 2*8+1+1)
                    + resize(s_sum0(2*i+1), 2*8+1+1);
            end loop;
      end if;
   end if;
end process p_sum1;

p_sum2: process(clock) is
begin
   if rising_edge(clock) then
      if data_valid = '1' then
         s_sum2 <= resize(s_sum1(0), 2*8+3)
           + resize(s_sum1(1), 2*8+3);
      end if;
   end if;
end process p_sum2;
```

```
  p_output: process(clock) is
  begin
     if rising_edge(clock) then
        if data_valid = '1' then
           -- we must lose precision in the 8bits output
           o_data <= std_logic_vector( shift_right(s_sum2, 6)(7 downto
               0) );
        end if;
     end if;
  end process p_output;
end architecture rtl;
```

## C.3 Top entity for the FIR Filter with 8 coefficients

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity top8 is
    port (
        CLK100MHZ   : in std_logic;
        uart_txd_in : in std_logic;
        uart_rxd_out : out std_logic
    );
end entity top8;

architecture str of top8 is
   signal data_to_send : std_logic_vector(7 downto 0) := X"00";
   signal data_valid : std_logic;
   signal busy       : std_logic;
   signal uart_tx    : std_logic;
   signal data_to_filter: std_logic_vector(7 downto 0) := X"00";

   component uart_receiver is
       port(
           clock        : in std_logic;
           uart_rx      : in std_logic;
           valid        : out std_logic;
           received_data : out std_logic_vector(7 downto 0));
   end component uart_receiver;

   component fir8 is
     port(
        -- clock
        clock: in std_logic;
        -- for resetting the values inside the entity
        data_valid: in std_logic;
        -- input data
        i_data: in std_logic_vector(7 downto 0);
        -- ouput (filtered) data
        o_data: out std_logic_vector(7 downto 0)
     );
   end component fir8;

    component uart_transmitter is
       port(
           clock       : in std_logic;
           data_to_send : in std_logic_vector(7 downto 0);
           data_valid  : in std_logic;
           busy        : out std_logic;
           uart_tx     : out std_logic);
```

```
   end component uart_transmitter;

begin -- architecture str

    uart_receiver_1 : uart_receiver
        port map(
            clock        => CLK100MHZ,
            uart_rx      => uart_txd_in,
            valid        => data_valid,
            received_data => data_to_filter
        );

   fir8_1: fir8
      port map(
         clock => CLK100MHZ,
         data_valid => data_valid,
         i_data => data_to_filter,
         o_data => data_to_send
   );

    uart_transmitter_1 : uart_transmitter
        port map(
            clock       => CLK100MHZ,
            data_to_send => data_to_send,
            data_valid  => data_valid,
            busy        => busy,
            uart_tx     => uart_rxd_out
        );

end architecture str;
```

## C.4 Testbench of the top entity for the FIR Filter with 8 coefficients

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;

entity top8_tb is
end entity top8_tb;

architecture tb of top8_tb is

    component top8 is
        port (
            CLK100MHZ   : in std_logic;
            uart_txd_in : in std_logic;
            uart_rxd_out : out std_logic
        );
    end component top8;
    signal CLK100MHZ: std_logic := '0';
    signal uart_txd_in: std_logic;
    signal uart_rxd_out: std_logic;

    signal clock_enable: std_logic := '1';
    file f_vectors: text;
   file f_results: text;
begin
    DUT: top8 port map(
        CLK100MHZ => CLK100MHZ,
        uart_txd_in => uart_txd_in,
        uart_rxd_out => uart_rxd_out
    );
    -- 10 ns period when enabling clock
    CLK100MHZ <= not CLK100MHZ after 5 ns when clock_enable = '1' else
         '0';

    process is
   variable counter: integer := 0;
   constant divisor: integer := 867;
    variable v_i_line, v_o_line: line;
    variable v_i_data_integer: integer := 0;
   variable v_o_data_integer: integer := 0;
    variable value_vec_in: std_logic_vector(7 downto 0) := X"00";
    variable value_vec_out: std_logic_vector(7 downto 0) := X"00";
    begin
        file_open(f_vectors, "input_vectors_reduced.txt", read_mode);
      file_open(f_results, "output_results8_top_tb.txt", write_mode);
        wait until rising_edge(CLK100MHZ);
```

```
  while not endfile(f_vectors) loop
   readline(f_vectors, v_i_line);
   read(v_i_line, v_i_data_integer);
   value_vec_in := std_logic_vector( to_signed(v_i_data_integer,
       8) );
   wait until rising_edge(CLK100MHZ);
   -- start bit
   uart_txd_in <= '0';
   while counter < divisor loop
      wait until rising_edge(CLK100MHZ);
      counter := counter + 1;
   end loop;
   counter := 0;
   -- value bits
   for i in 0 to 7 loop
      while counter < divisor loop
         wait until rising_edge(CLK100MHZ);
         uart_txd_in <= value_vec_in(i);
         counter := counter + 1;
      end loop;
      counter := 0;
   end loop;
   -- stop bit
   uart_txd_in <= '1';
   wait until rising_edge(CLK100MHZ);
   -- start bit
   wait until uart_rxd_out = '0';
   while counter < divisor loop
      wait until rising_edge(CLK100MHZ);
      counter := counter + 1;
   end loop;
   counter := 0;
   -- value bits
      for i in 0 to 7 loop
      while counter < divisor loop
         wait until rising_edge(CLK100MHZ);
         value_vec_out(i) := uart_rxd_out;
         counter := counter + 1;
      end loop;
      counter := 0;
   end loop;
   -- writing to file
   v_o_data_integer := to_integer(signed(value_vec_out));
   write(v_o_line, v_o_data_integer, left, 8);
   writeline(f_results, v_o_line);
end loop;

file_close(f_vectors);
file_close(f_results);
clock_enable <= '0';
```

```
    wait;
  end process;
end architecture tb;
```