INTRODUCTION

This tutorial presents a comprehensive, hands-on guide to designing and implementing an **FSK (Frequency Shift Keying) modem** on the **ARTY A7 FPGA board**. Using **Simulink**, **Vitis HLS**, and **Vivado**, the tutorial demonstrates the process of building a functional digital communication system from simulation to hardware, including testbenches, filtering, modulation/demodulation, and UART integration. It is intended for engineers and students looking to understand practical aspects of digital modem implementation on FPGAs.

Summary

The document is divided in the following main sections:

1. FSK Modulation and Demodulation

- Introduces FSK principles and asynchronous detection using envelope detectors with squaring and low-pass filtering.
- Modulation and demodulation blocks are modelled in Simulink with 10 kHz and 15 kHz carriers at a 1 MHz sampling rate.
- A testbench in MATLAB runs the simulation and verifies the waveforms, forming the basis for FPGA implementation.

2. HLS Filter Design

- Filters from the Simulink model are re-implemented in C/C++ using **Vitis HLS** (High-Level Synthesis).
- Focuses on second-order **direct form II** filters, implemented via C code based on Simulink coefficients.
- Includes validation using GDB-based testbenches and waveform analysis.
- The HLS filters (two bandpass and one low-pass) are compiled into IP-cores for use in Vivado.

3. The Vivado Project

- Managed through bash and TCL scripts for automation.
- Uses a **block design** to graphically instantiate modem components, including the HLS filters.
- Additional VHDL modules (comparator, 2:1 MUX, sinewave generator) support core modem functionality.

- A VHDL testbench simulates binary modulation-demodulation using a pseudo-random binary source.

4. The Vivado UART Testbench

- Integrates UART transmission with FSK modulation, creating a robust test scenario for real-world communication.
- Applications span embedded, industrial, automotive, satellite, and SCADA systems.
- Simulates a complete UART transmission path through an FSK modem and back into a UART receiver.
- Verifies data integrity by looping 1024 transmissions and confirming that transmitted and received data match modulo 256.
- Demonstrates system stability and correctness even under multiple transfers.

FSK MODULATION AND DEMODULATION

A description about FSK digital modulation is available at:

https://www.tutorialspoint.com/digital_communication/digital_communication_frequency_shift_keying.htm

And:

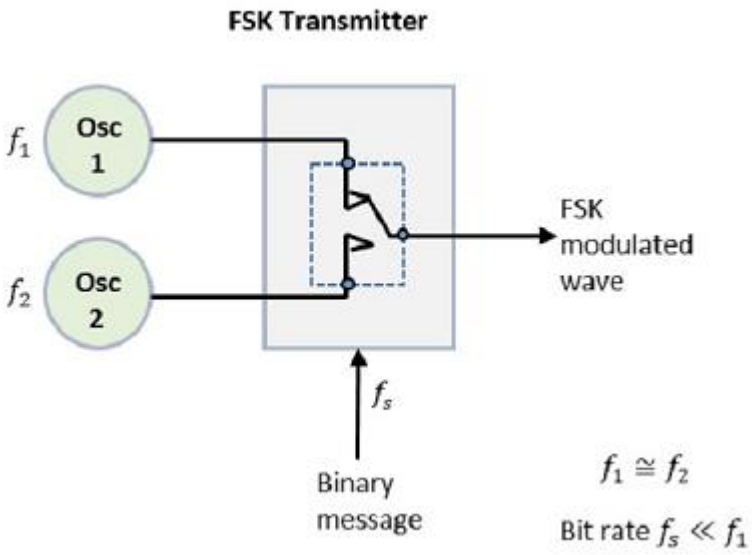
https://www.technologyuk.net/telecommunications/telecom-principles/digital-modulation-part-one.shtml

We are going now to do a preliminary Simulink model with the modulation and demodulation blocks, to prove the feasibility of the design we are going to create for the FPGA.

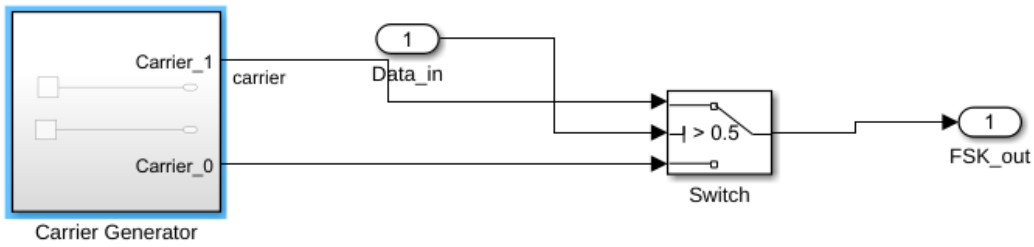
Asynchronous detection is used, and envelope detectors with squaring and lowpass filtering will be used in the receiver:

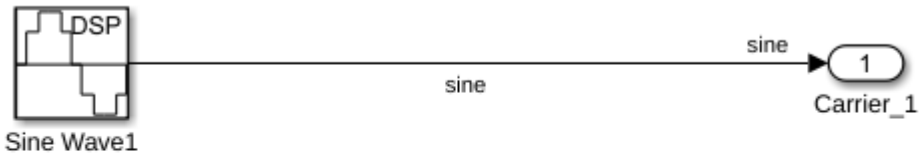
https://uk.mathworks.com/help/dsp/ug/envelope-detection-in-simulink.html

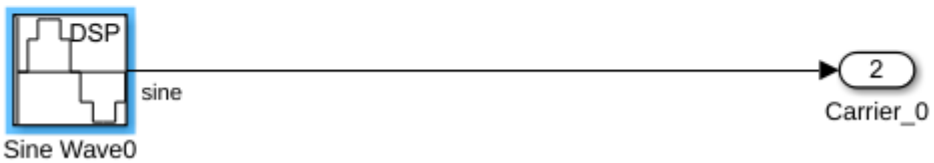
The modulator scheme:



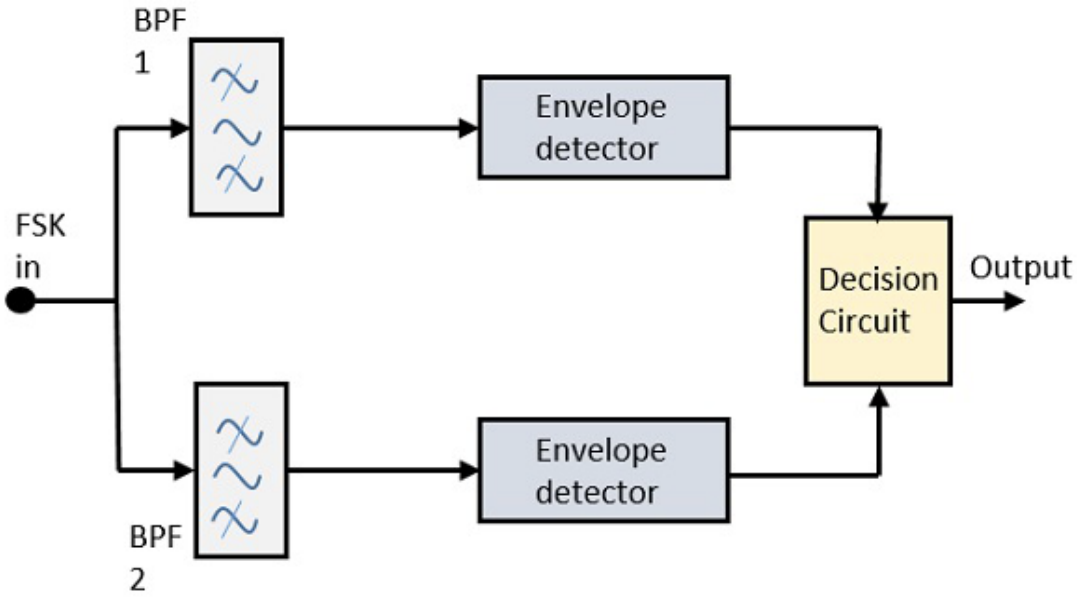
Has been done in Simulink:



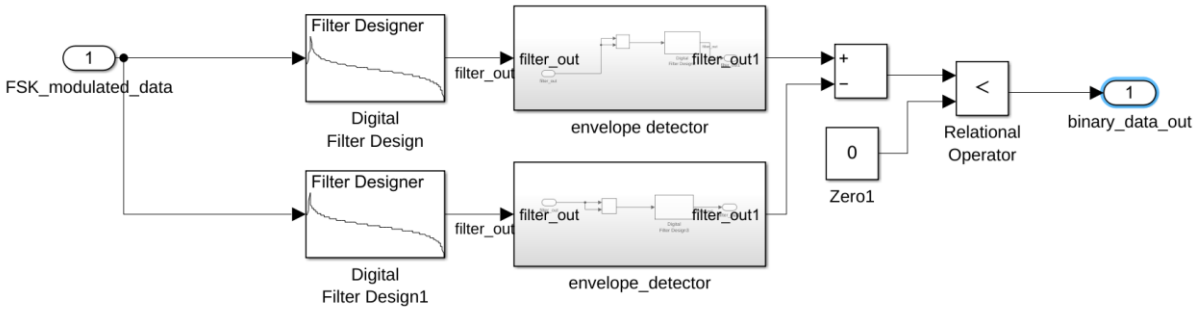




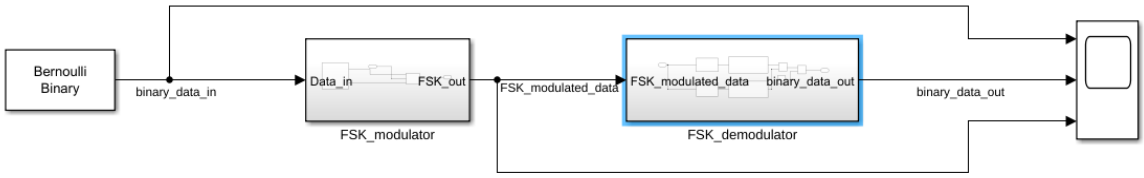
Similarly, the demodulator scheme:



Has been reproduced in simulink as well:



To test them, the following simulink diagram has been prepared:



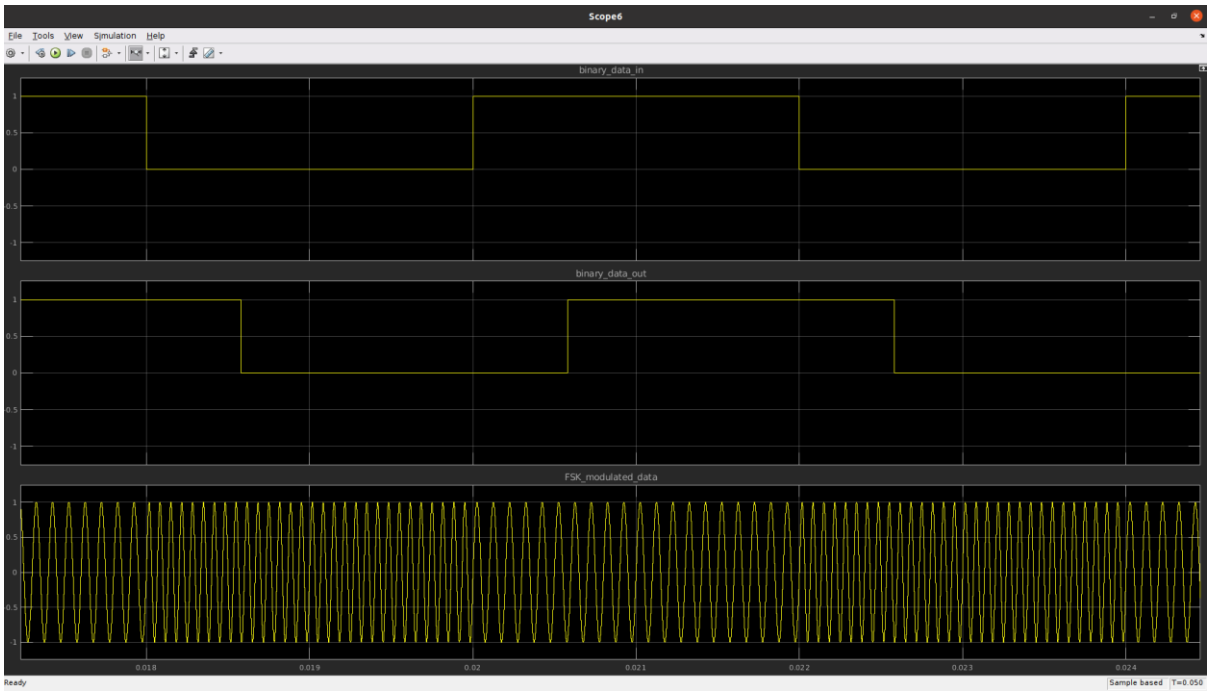
Aided by a matlab script used to set up and run the simulation:

Editor - /home/caccolillo/Downloads/FSK_MODEM/SIMULINK/run_modem.m

run_modem.m

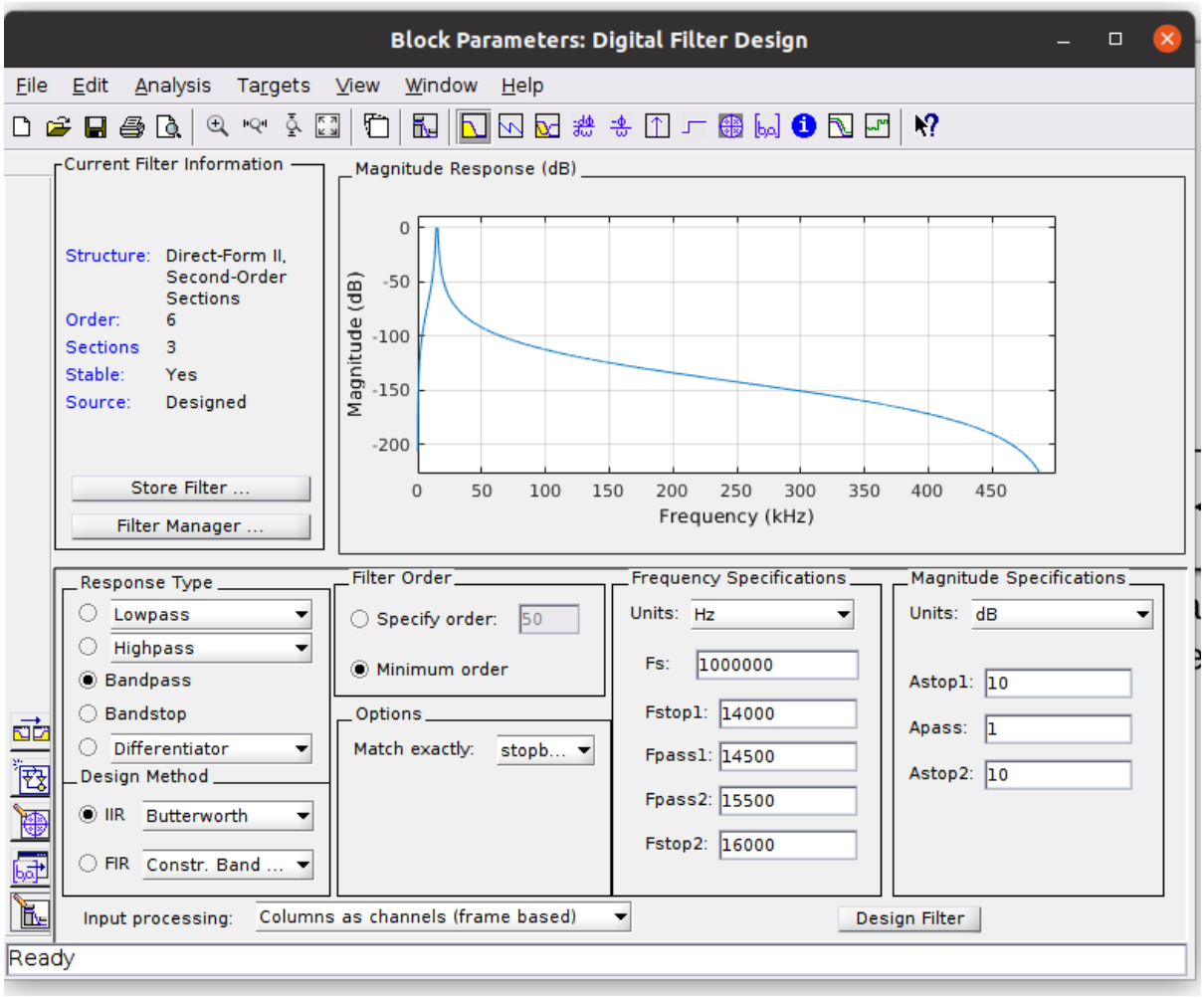
```
% MATLAB Script to open

% Define the model name
modelName = 'FSK_MODEM';

fcarrier = 10000;
fdelta = (fcarrier/100)*50;

sample_time = 1/(100*fcarrier);

% Check if the model is already loaded
if ~bdIsLoaded(modelName)
    % Load the model
    load_system(modelName);
end

% Open the model in Simulink
open_system(modelName);

% Run the simulation
sim(modelName);

% Optional: Display message when simulation is complete
disp('Simulation complete.');
```

Key waveforms have been plot, to confirm the validity of the model visually:

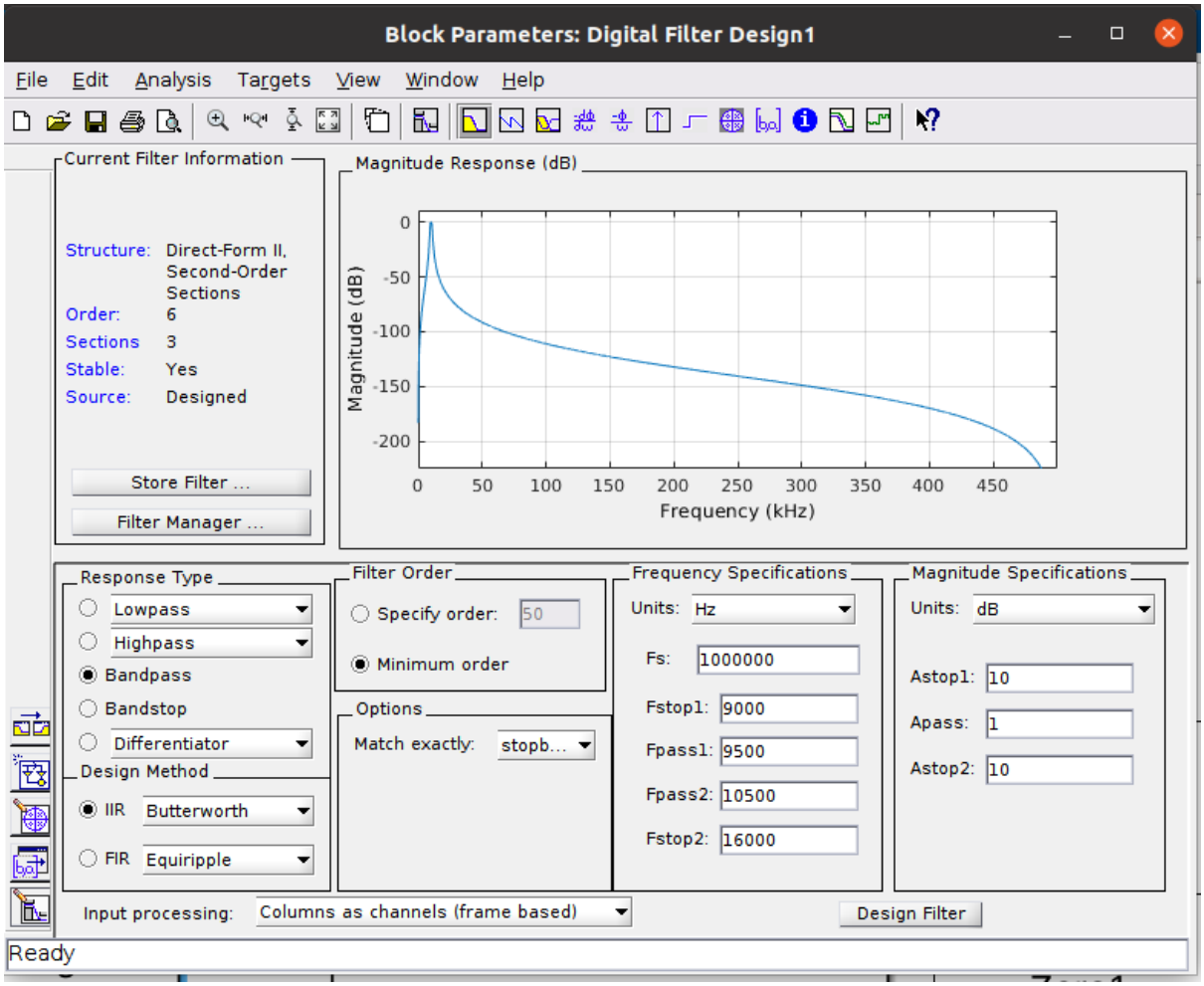


The system is using a 1 MHz sampling rate, we are using a 10 KHz and 15 KHz carriers.

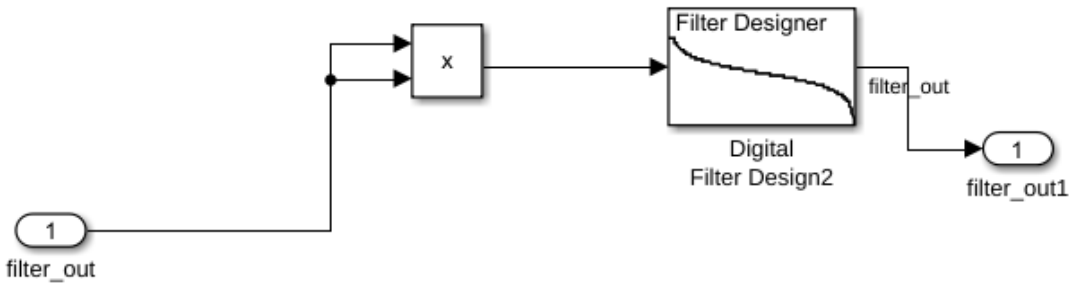
The BPF filters in the receiver are configured as follows:

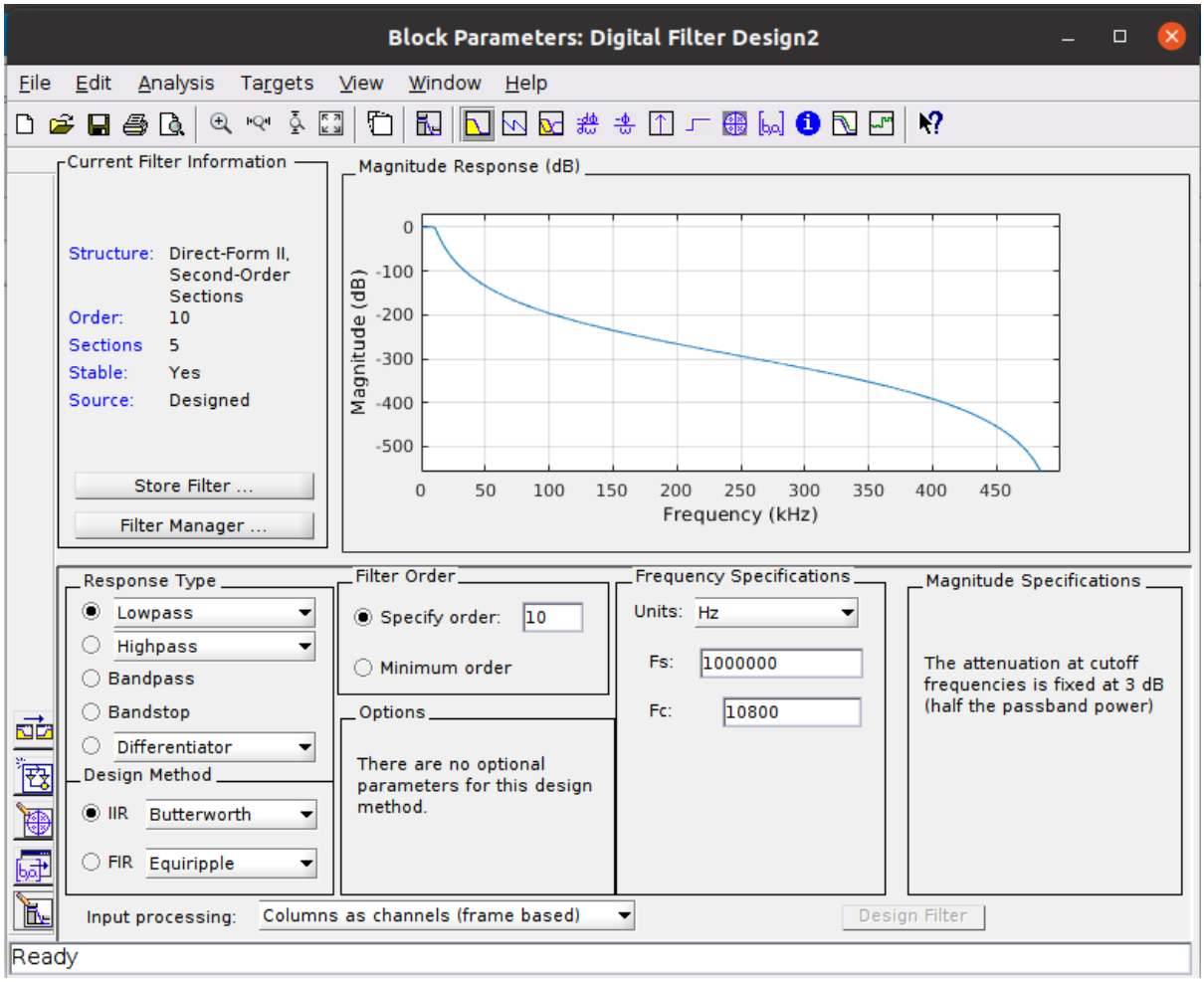


The simulink model, will be used as a blueprint to design the FPGA modem hereon.

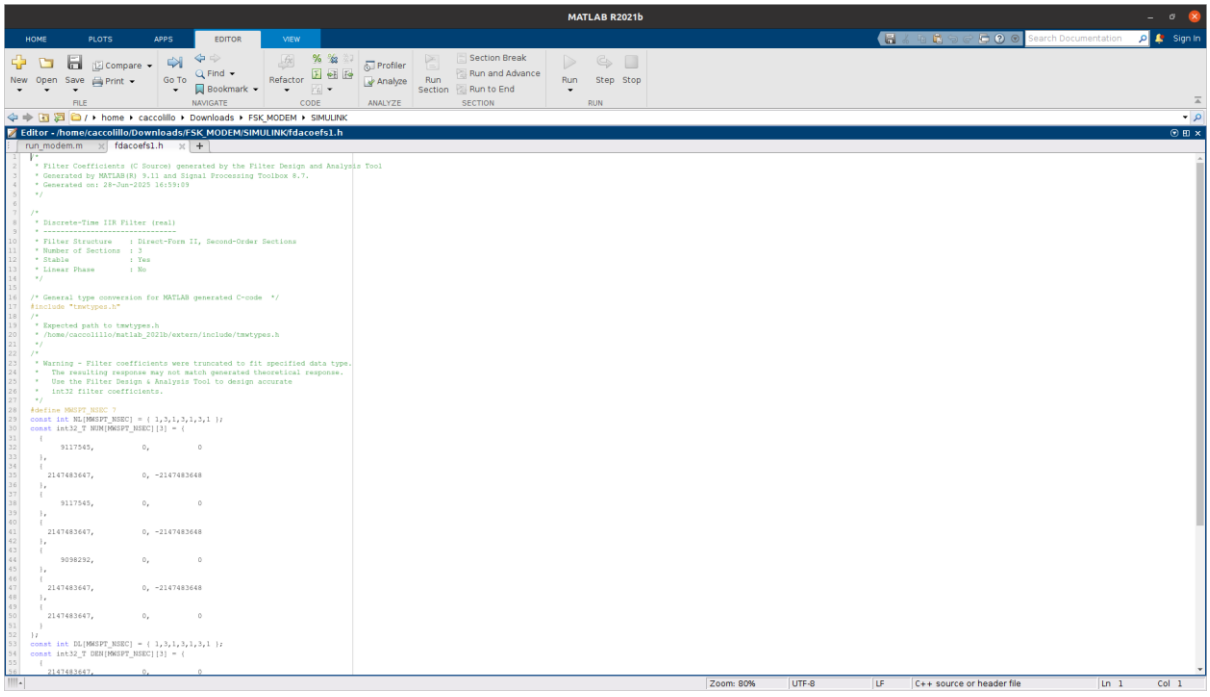


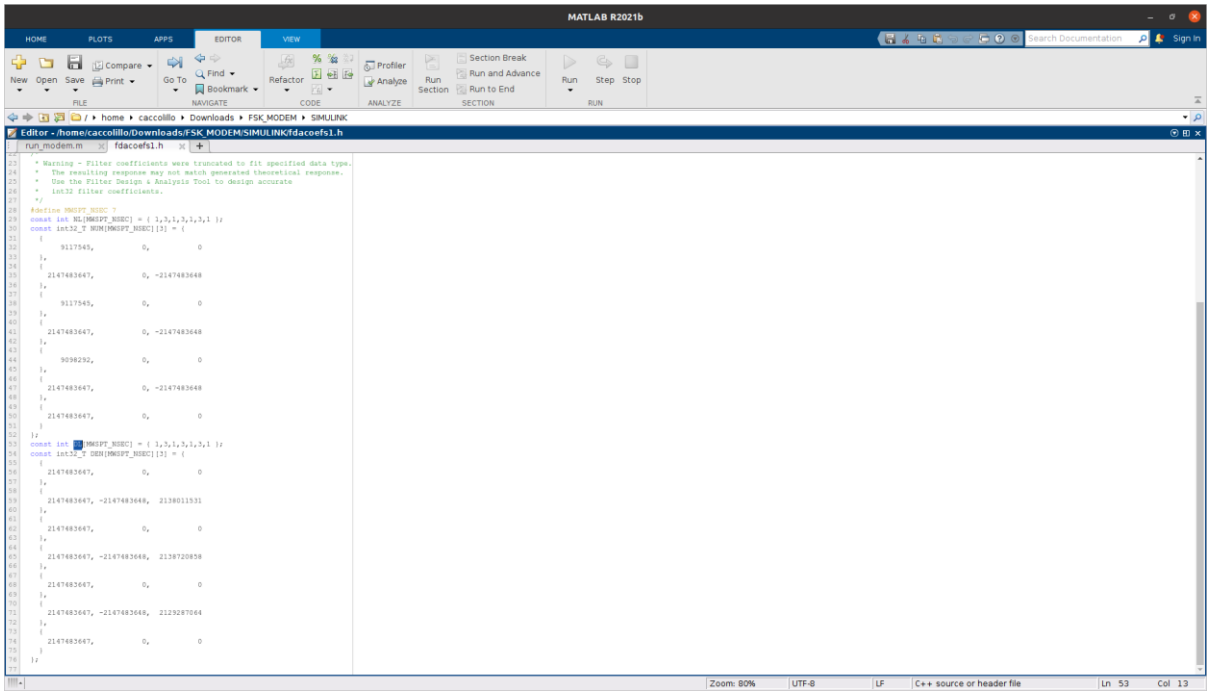
The envelope detector has been implemented as shown below:





For each filter, we can get a C header file with coefficients:





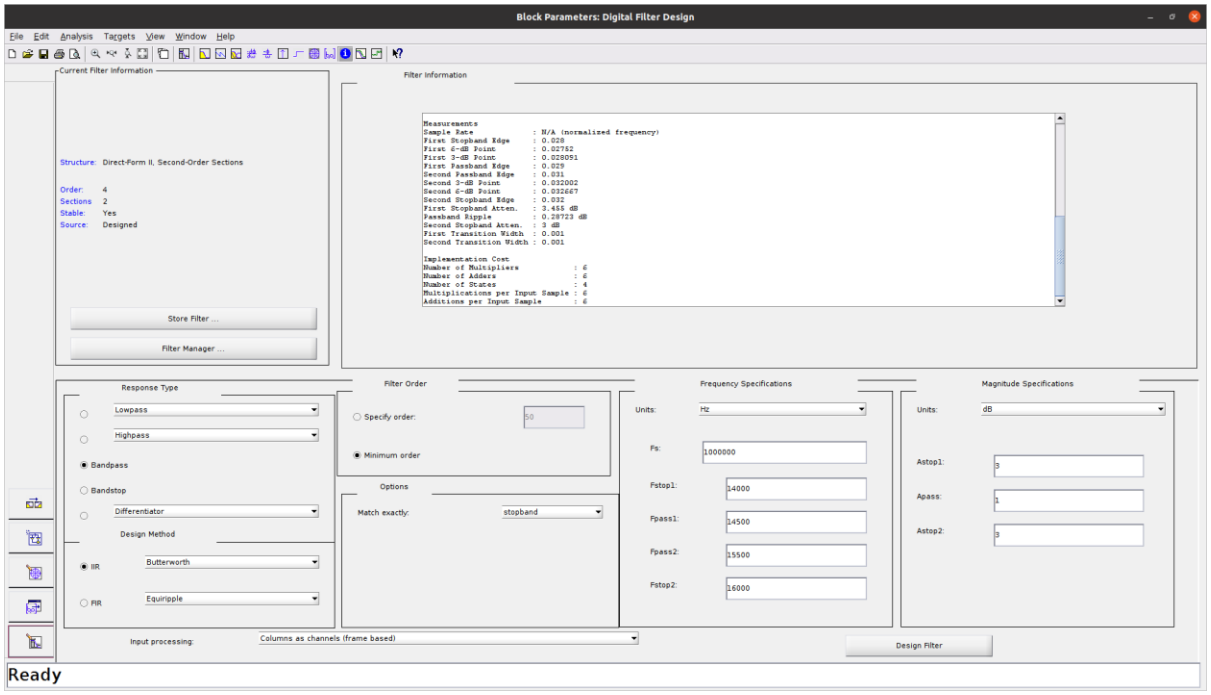
It will be the starting point to implement the filters via Vitis HSL.

HLS FILTER

HLS allows the design of filters seen in the Simulink model, by using C/C++ hardware aware code.

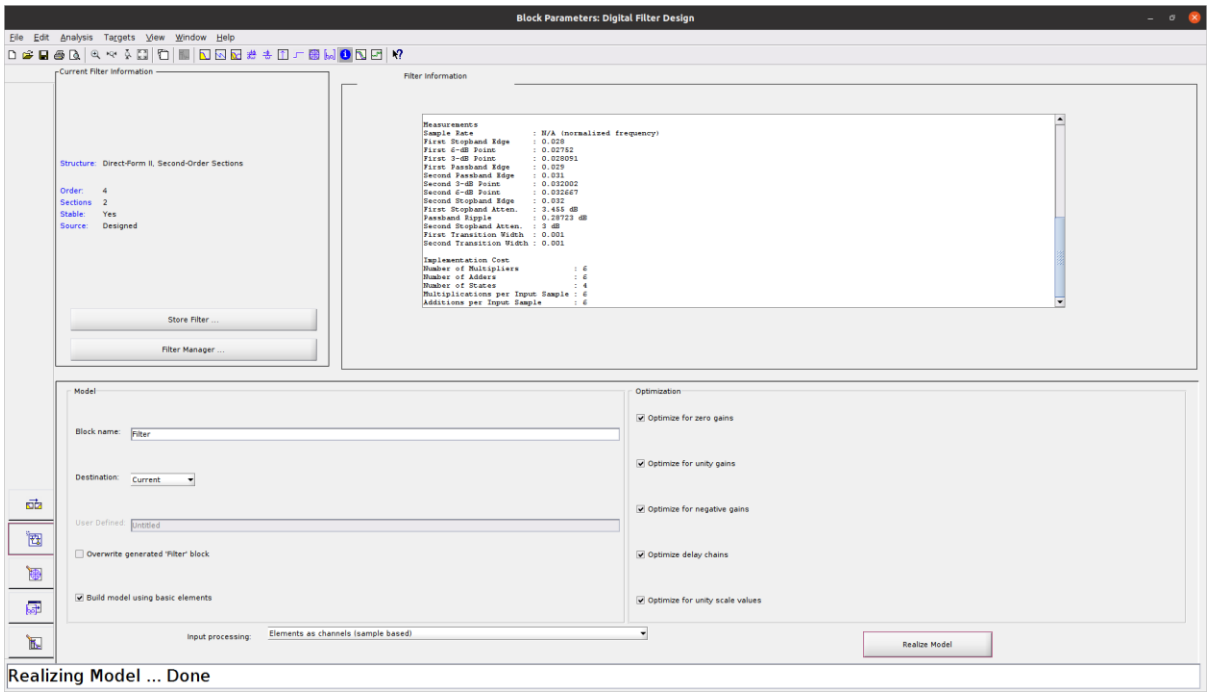
Now let's delve into the details of the filters.

By opening the first BPF, we can have a look at its HW occupancy report:

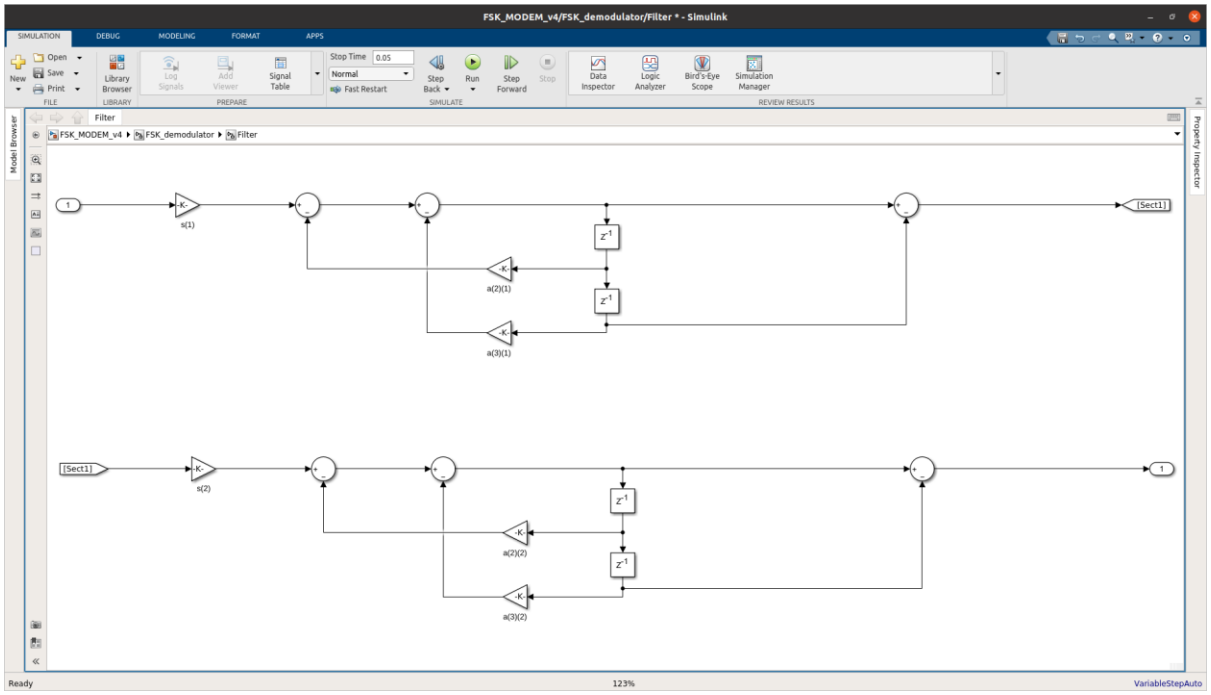


There's a summary about the number of arithmetic resources used, namely adders, multipliers, alongside the number of delay lines.

By pressing on the model symbol on the bottom of the left panel, and by selecting “build model using basic elements”, and then by pressing “realise model”:



A block diagram with the filter gets created:



The filter topology is direct form II, second order sections.

For some theory background, the following resources are useful:

https://www.dsprelated.com/freebooks/filters/Four_Direct_Forms.html

https://www.dsprelated.com/freebooks/filters/Direct_Form_II.html

https://www.ti.com/lit/an/spra509/spra509.pdf?ts=1754211231378&ref_url=https%253A%252F%252Fwww.google.com%252F

The block diagram above, can be used as a source of inspiration to code in C the filter to be implemented in HLS.

The landing page from where gather information about HLS is:

https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-hls.html

For the absolute beginner, the tutorial might be a good idea:

https://docs.amd.com/r/en-US/Vitis-Tutorials-Getting-Started/Vitis-HLS

Going back to the C code, the filter gets coded as follows:

Synthesis Summary(solution1) filter1.cpp

```cpp
#include "filter1.hpp"


// Section 1: static internal state
data_type section1(data_type input) {
    static bool initialized = false;
    static data_type d1;
    static data_type d2;

    // One-time initialization
    if (!initialized) {
        d1 = 0.0;
        d2 = 0.0;
        initialized = true;
    }


    // Scaled input
    accum_type input_scaled = input * scaleconst1_section1;

    // Filter math
    accum_type a2mul = d1 * coeff_a2_section1;
    accum_type a3mul = d2 * coeff_a3_section1;
    accum_type a1sum = (input_scaled - a2mul) - a3mul;
    accum_type b1sum = a1sum - d2;  // b3 = -1 * d2

    // Update delay line
    d2 = d1;
    d1 = a1sum;

    // Scaled output
    return b1sum * scaleconst2_section1;
}

// Section 2: static internal state
data_type section2(data_type input) {
    static bool initialized = false;
    static data_type d1;
    static data_type d2;

    // One-time initialization
    if (!initialized) {
        d1 = 0.0;
        d2 = 0.0;
        initialized = true;
    }

    // Filter math
    accum_type a2mul = d1 * coeff_a2_section2;
    accum_type a3mul = d2 * coeff_a3_section2;
    accum_type a1sum = (input - a2mul) - a3mul;
    accum_type b1sum = a1sum - d2;  // b3 = -1 * d2

    // Update delay line
    d2 = d1;
    d1 = a1sum;

    return b1sum;
}

// Top-level function using both sections
data_type filter1(data_type input) {
    data_type stage1;
    data_type output;

    stage1 = section1(input);
    output = section2(stage1);
    return output;
}
```

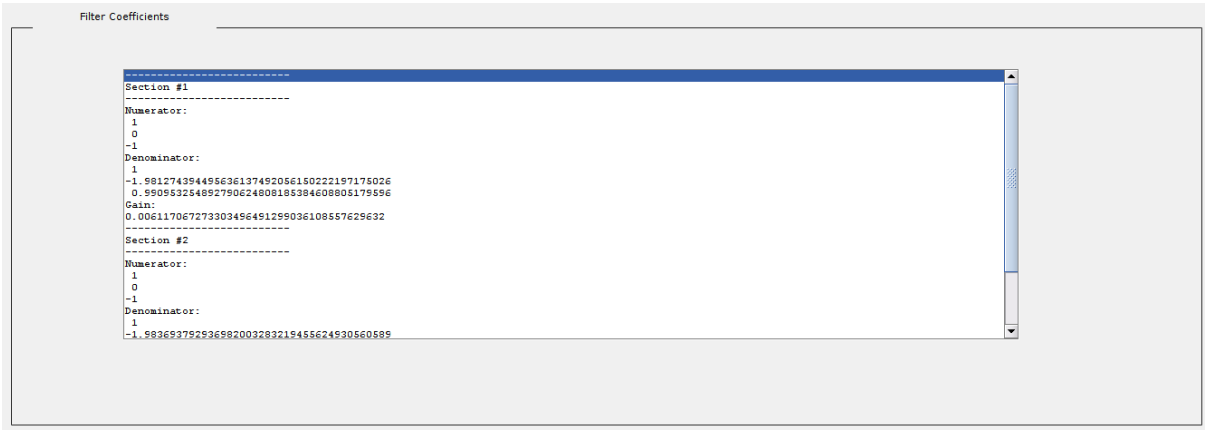
Filter coefficients have been saved in the header file:

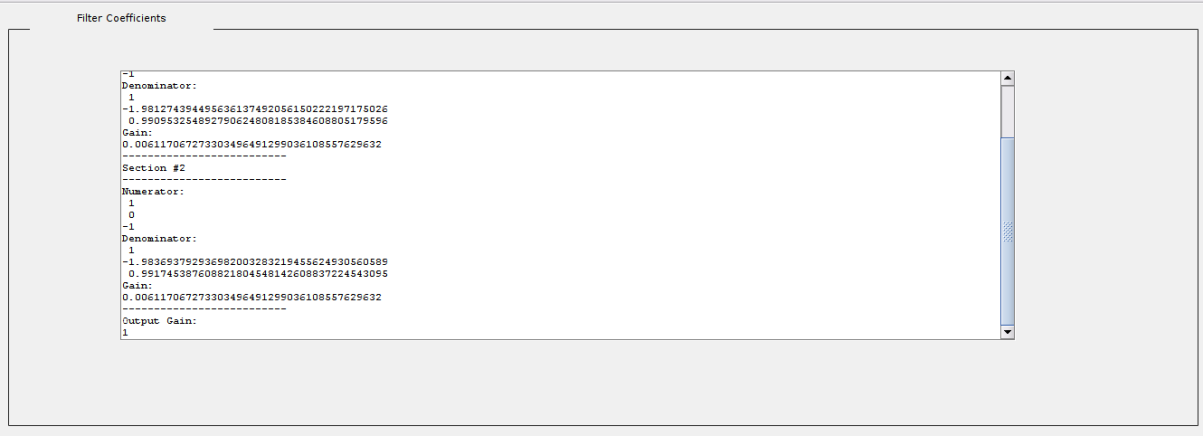
Synthesis Summary(solution1) filter1.cpp filter1.hpp

```
#include <ap_int.h>
#include <ap_fixed.h>
#include <math.h>


// Data format
const int DataWordSize = 25;
const int DataIntSize = 3;
const float DataMaxVal = pow(2.0, DataIntSize-1);
typedef ap_fixed<DataWordSize, DataIntSize, AP_RND, AP_SAT, 0> data_type;


//coefficient constants and data types
const int CoeffWordSize = 18;
const int CoeffIntSize = 4;
typedef ap_fixed<CoeffWordSize, CoeffIntSize, AP_RND, AP_SAT, 0> coeff_type;



// Coefficients for section 1
const coeff_type scaleconst1_section1 = 6.1170672733034965E-03;
const coeff_type coeff_a2_section1 = -1.9812743944956361E+00;
const coeff_type coeff_a3_section1 = 9.9095325489279062E-01;
const coeff_type scaleconst2_section1 = 6.1170672733034965E-03;

// Coefficients for section 2
const coeff_type coeff_a2_section2 = -1.9836937929369820E+00;
const coeff_type coeff_a3_section2 = 9.9174538760882180E-01;

typedef ap_fixed<DataWordSize+CoeffWordSize+3, DataIntSize+CoeffIntSize, AP_TRN, AP_WRAP, 0> accum_type;



// Processes one sample through the fixed-point IIR filter
data_type filter1(data_type input);
data_type section1(data_type input);
data_type section2(data_type input);
```

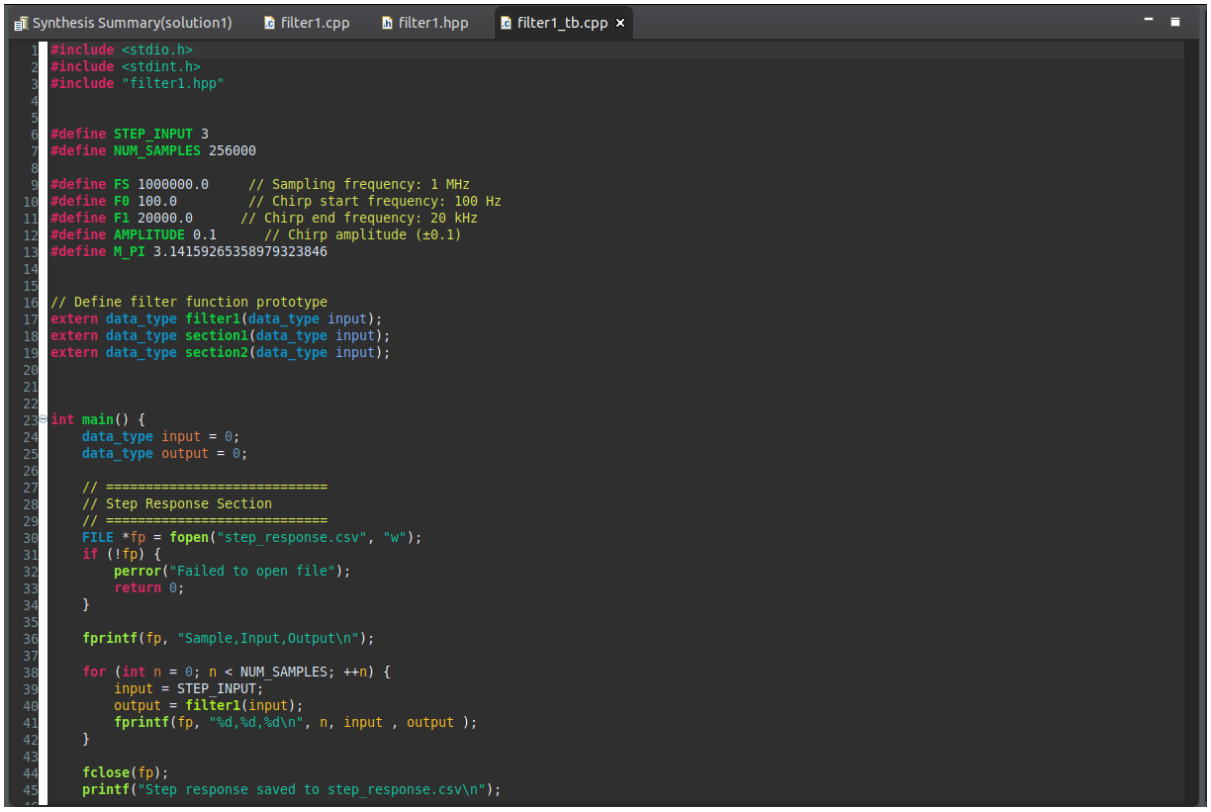
Fixed point arithmetic is in use to perform arithmetic operations.

Coefficients have been taken from Simulink, as can be seen from the excerpt of the Simulink filter designer coefficient section:



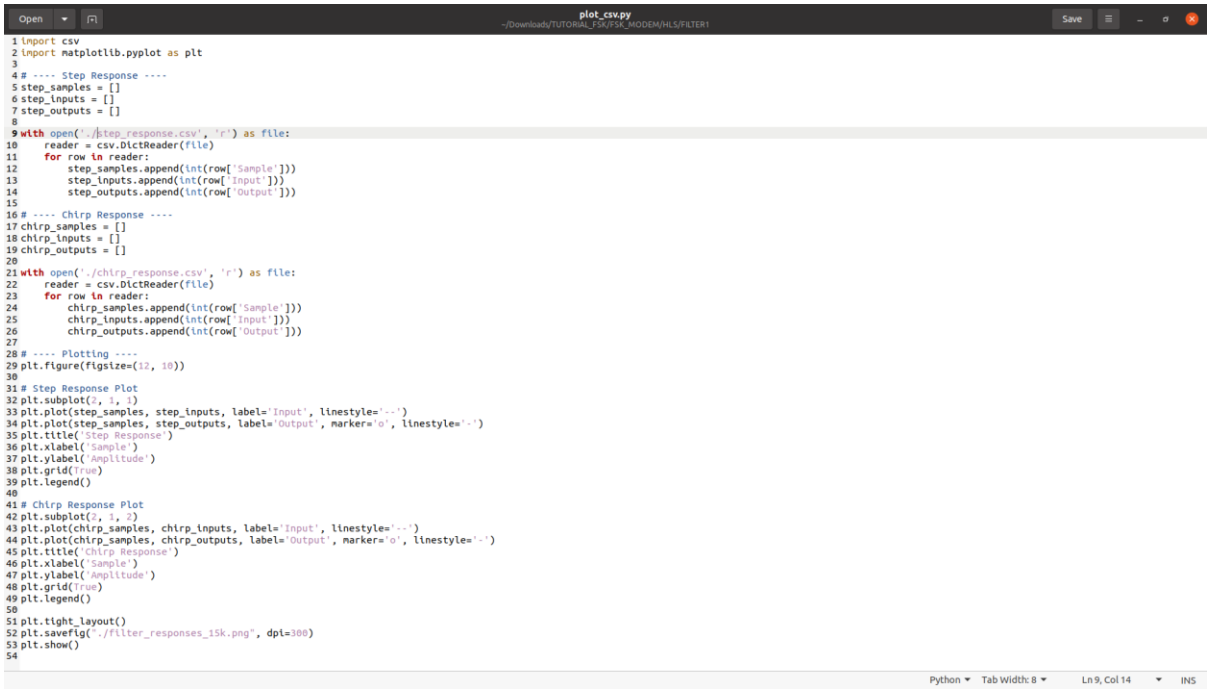


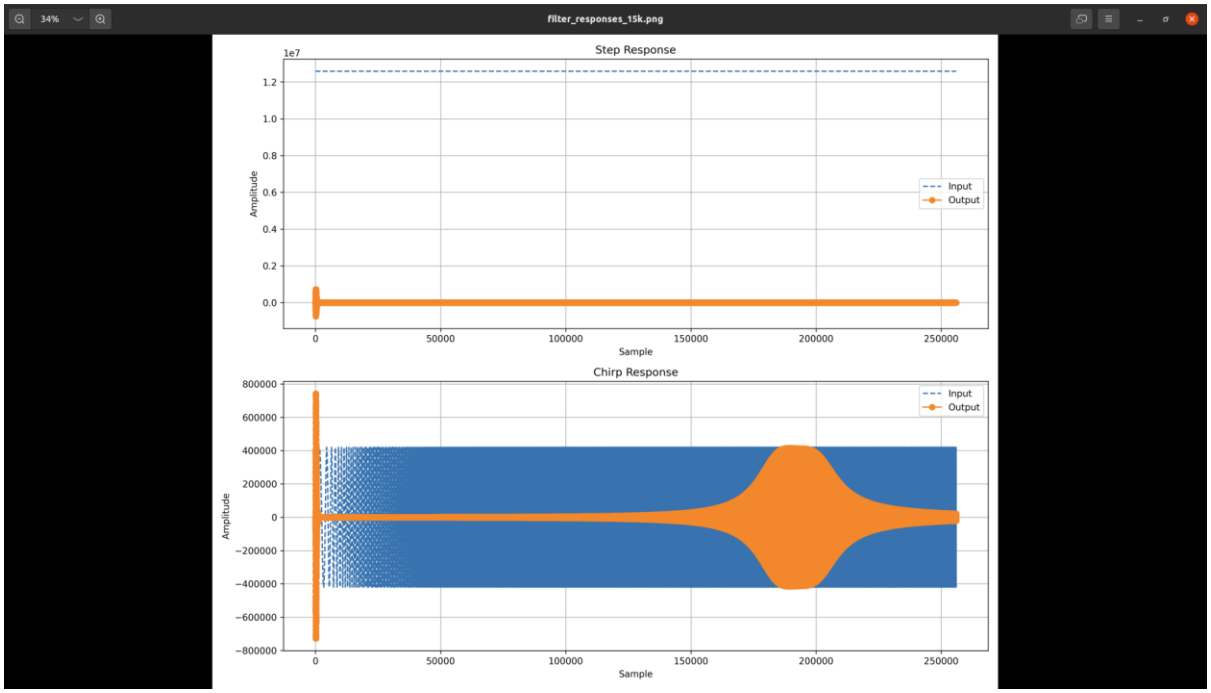
A C testbench has been prepared for an early validation of the filter by using GDB:



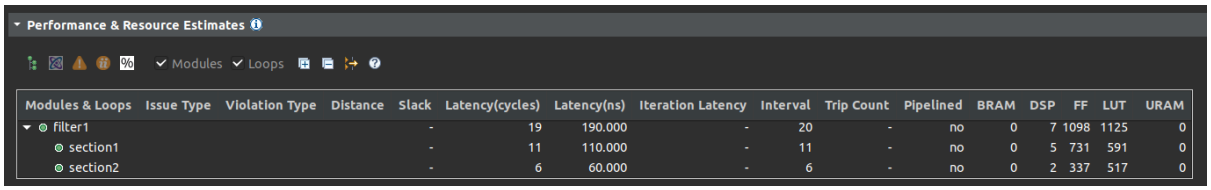
```
    printf("Step response saved to step_response.csv\n");

    // ============================
    // Chirp Signal Section
    // ============================
    FILE *fp_chirp = fopen("chirp_response.csv", "w");
    if (!fp_chirp) {
        perror("Failed to open chirp_response.csv");
        return 0;
    }

    fprintf(fp_chirp, "Sample,Input,Output\n");

    double T = NUM_SAMPLES / FS;         // Total time duration (in seconds)
    double k = (F1 - F0) / T;            // Chirp rate (Hz/s)

    for (int n = 0; n < NUM_SAMPLES; ++n) {
        double t = n / FS;               // Time in seconds
        double phase = 2.0 * M_PI * (F0 * t + 0.5 * k * t * t);  // Chirp phase
        double chirp_val = AMPLITUDE * sin(phase);              // Floating-point value in [-1,1]

        // Convert to signed integer in range [-1, 1]
        input = (data_type)(chirp_val);
        output = filter1(input);

        fprintf(fp_chirp, "%d,%d,%d\n", n, input, output);
    }

    fclose(fp_chirp);
    printf("Chirp response saved to chirp_response.csv\n");

    // Run Python plotting script
    system("python3 ./plot_csv.py");

    return 0;
}
```

The testbench collects the response of the filter to an input step signal, and a chirp, collects data in a csv file, and in the end runs a python script used to plot the csv files:





In terms of occupied resources, we have:



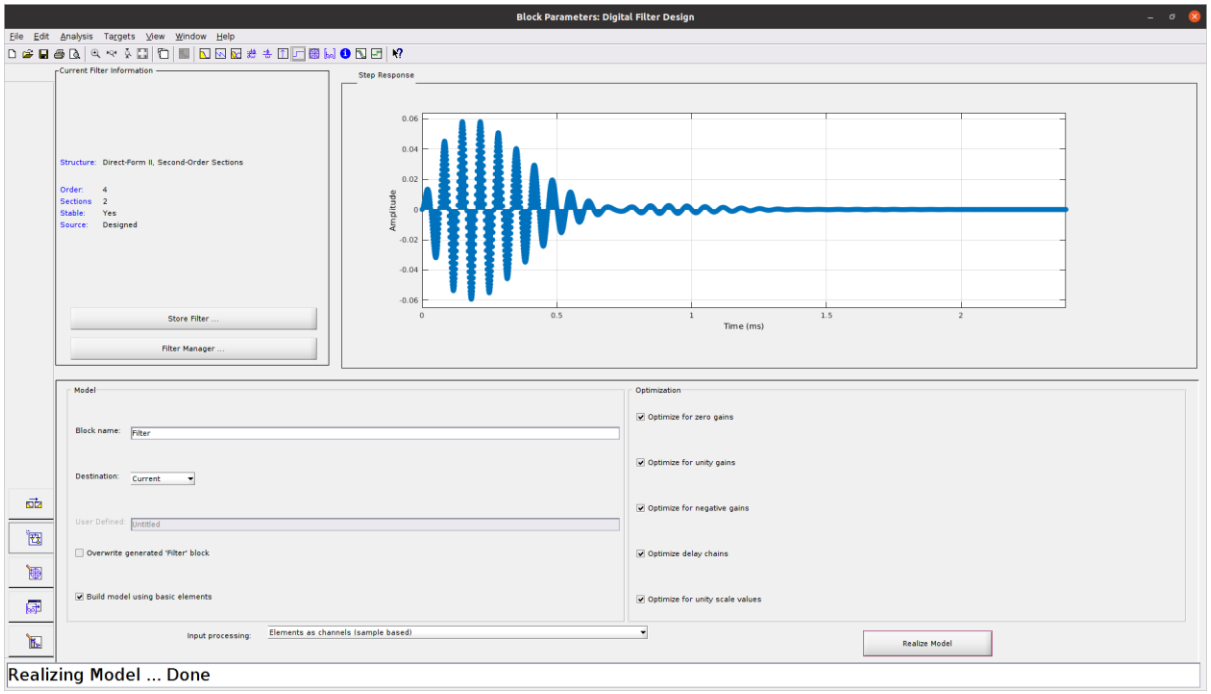
Performance & Resource Estimates

Modules Loops

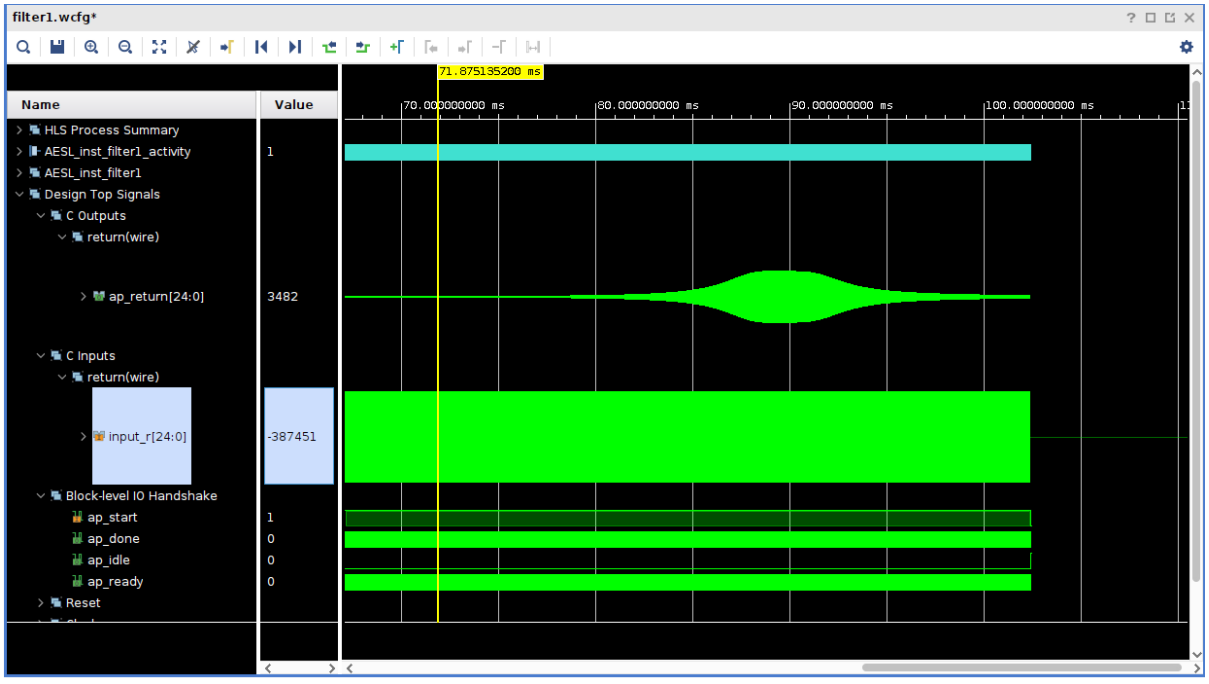
Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
filter1			-	-	19	190.000	-	20	-	no	0	7	1098	1125	0
section1			-	-	11	110.000	-	11	-	no	0	5	731	591	0
section2			-	-	6	60.000	-	6	-	no	0	2	337	517	0

The very same testbench seen before, can also be used to perform the C to RTL simulation of the implemented HW. Once executed it, by opening the waveform viewer, we can see the HW filter implementation in action:

The step response closely resembles the one of the starting filter used in Simulink:



We can also look at the response of the filter to a chirp input signal:



Once tested, the filter gets exported as an IP-core, to be included in Vivado's IP-catalogue.

All the project is automatically managed by bash and tcl scripts:

hls_build.sh
~/Downloads/TUTORIAL_FSK/FSK_MODEM/HLS/FILTER1

plot_csv.py | hls_build.sh

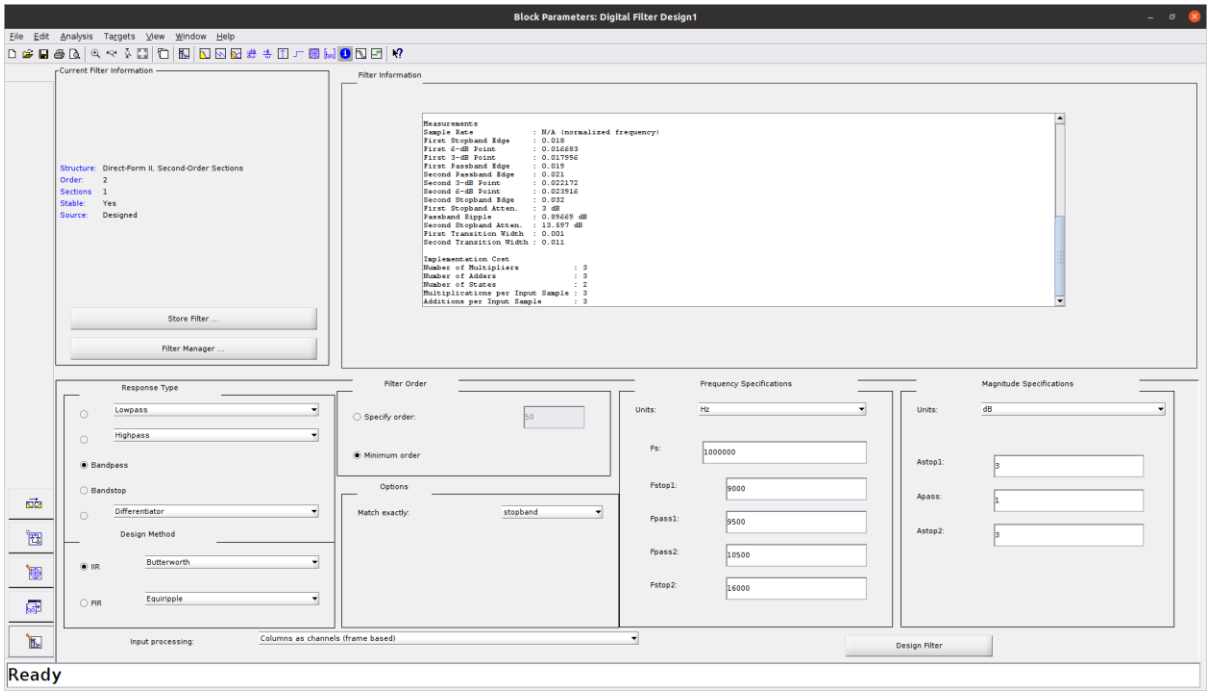
```
#!/bin/bash

# Script name: run_vitis_hls.sh

# Exit on any error
set -e

echo "Starting Vitis HLS build process..."

# Run the first Vitis HLS command
echo "Running: vitis_hls hls_build.tcl"
vitis_hls hls_build.tcl

# Run the second Vitis HLS command
#echo "Running: vitis_hls -p filter_envelope_detector/hls.app"
#vitis_hls -p filter_envelope_detector/hls.app

echo "Vitis HLS build completed successfully."

```

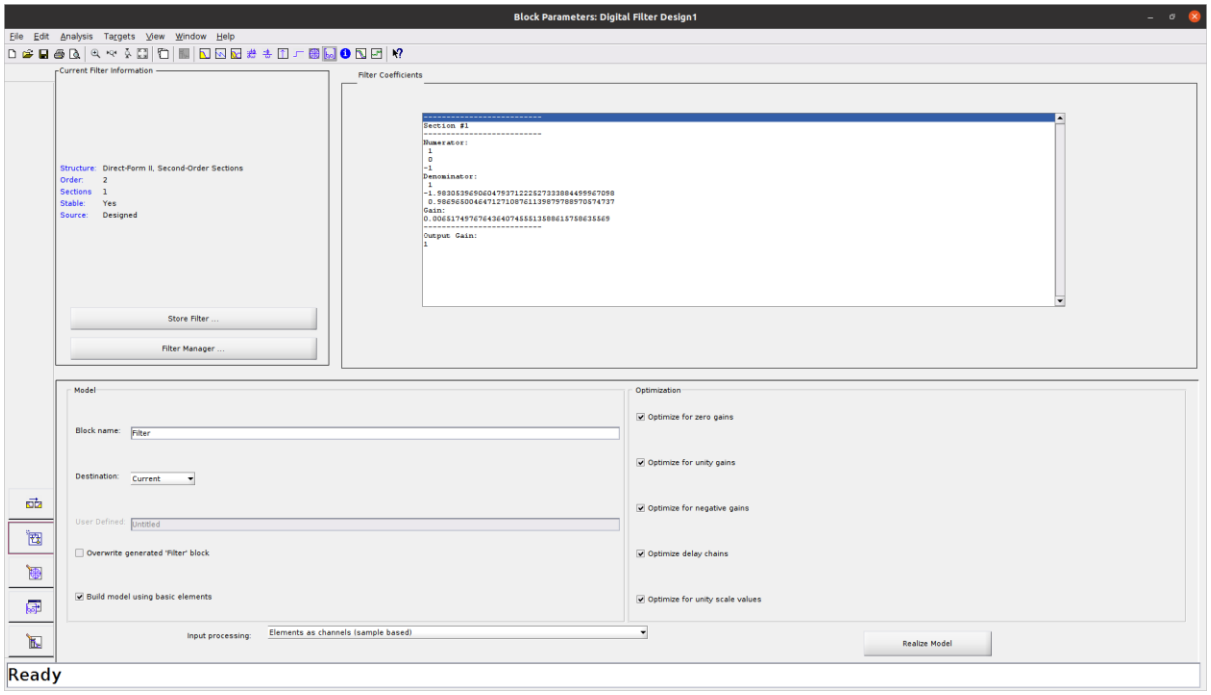
hls_build.tcl
~/Downloads/TUTORIAL_FSK/FSK_MODEM/HLS/FILTER1

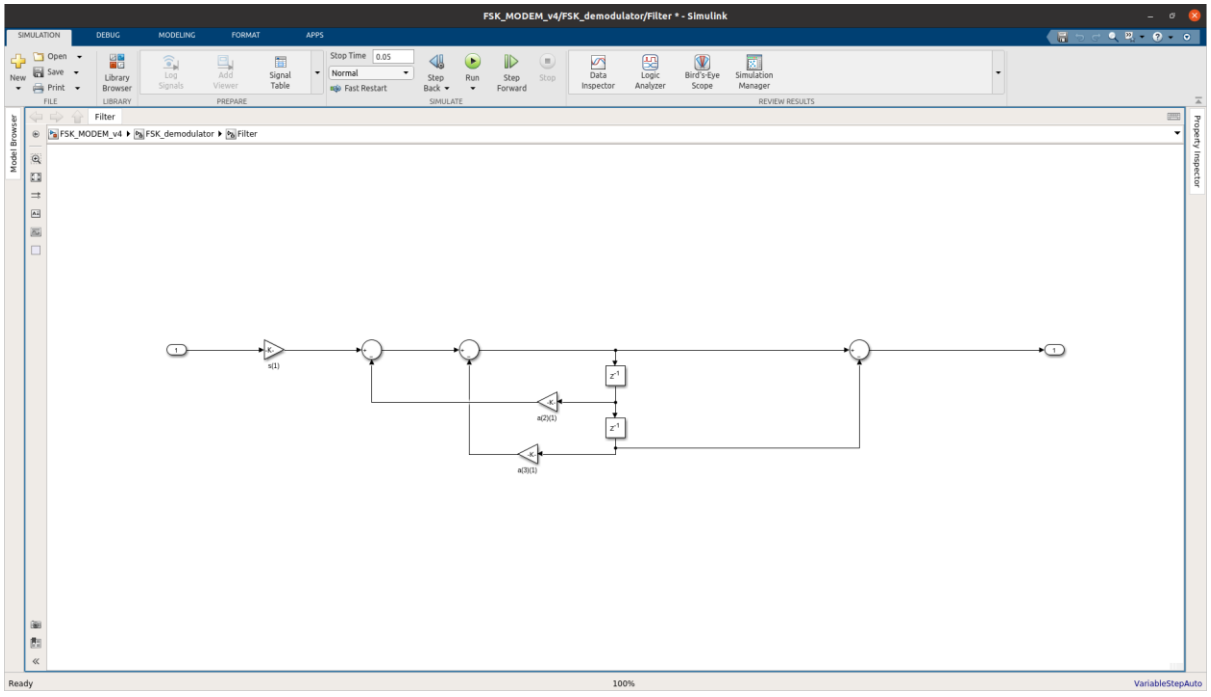
plot_csv.py | hls_build.sh | hls_build.tcl

```
# This script sets up and compiles an HLS project.
#
# vitis_hls hls_build.tcl
# vitis_hls -p hls_proj/hls.app

open_project -reset band_pass_filter_15k
set_top filter1
add_files filter1.hpp
add_files filter1.cpp
add_files -tb filter1_tb.cpp
add_files -tb plot_csv.py
open_solution -reset "solution1"
set_part {xc7a100t-csg324-1}
create_clock -period 10 -name default
config_export -format ip_catalog -rtl verilog
csynth_design
csim_design -clean
cosim_design
export_design -rtl verilog -format ip_catalog -description "BPF_15KHz" -vendor "caccolillo" -display_name "filter1" -output "../filter1.zip"
exit
```

A similar approach has been followed for the second bandpass filter:




Block Parameters: Digital Filter Design1
File Edit Analysis Targets View Window Help
Current Filter Information
Structure: Direct-Form II, Second-Order Sections
Order: 2
Sections: 1
Stable: Yes
Source: Designed
Store Filter ...
Filter Manager ...
Filter Coefficients
Section #1
Numerator:
1
0
-1
Denominator:
1
-1.9830639690834793712228273338844959670988
0.9869680046471271087611398797889570974737
Gain:
0.0065174976764364074555135886187588635569
Output Gain:
1
Model
Block name: Filter
Destination: Current
User Defined: Untitled
Overwrite generated 'Filter' block
Build model using basic elements
Optimization
Optimize for zero gains
Optimize for unity gains
Optimize for negative gains
Optimize delay chains
Optimize for unity scale values
Input processing: Elements as channels (sample based)
Realize Model
Ready

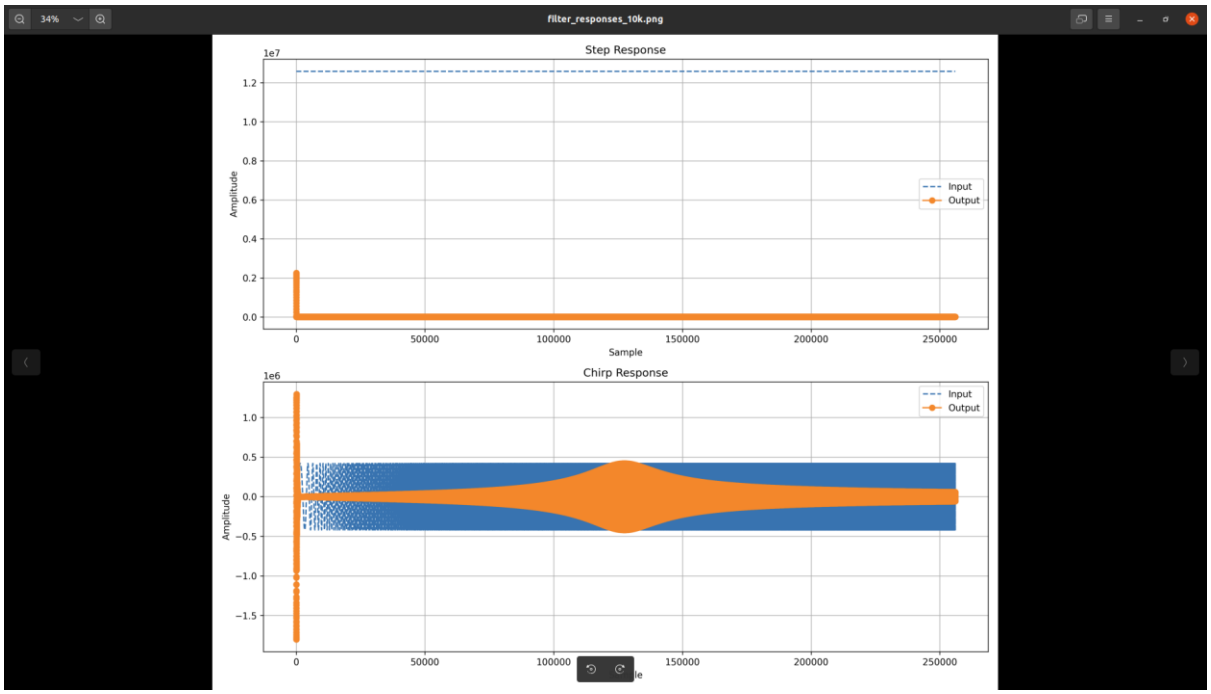

FSK_MODEM_v4/FSK_demodulator/Filter * - Simulink
SIMULATION DEBUG MODELING FORMAT APPS
New Open Save Print
Library Browser
Log Signals Add Viewer Signal Table
Stop Time 0.05
Normal
Fast Restart
Step Back Run Step Forward Stop
Data Inspector Logic Analyzer Bird's-Eye Scope Simulation Manager
FILE LIBRARY PREPARE SIMULATE REVIEW RESULTS
Model Browser
Filter
FSK_MODEM_v4 FSK_demodulator Filter
Property Inspector
s(1)
a(2)(1)
a(3)(1)
Ready
100%
VariableStepAuto

Synthesis Summary(solution1) filter2.cpp

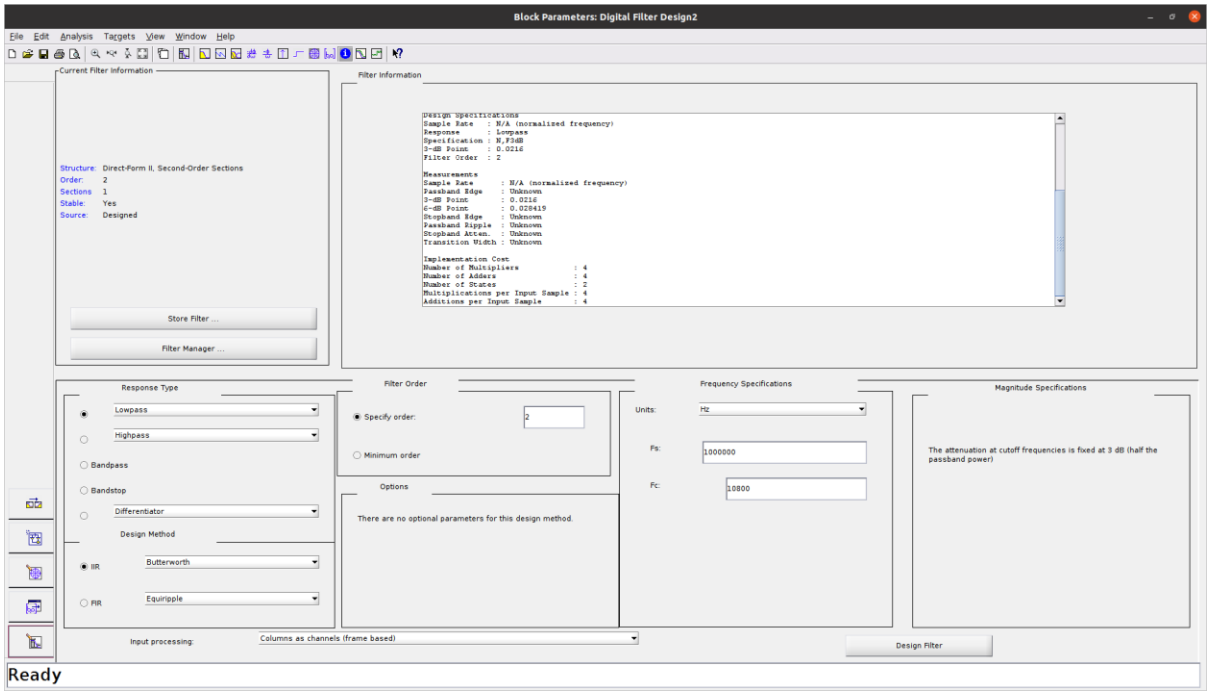
```cpp
#include "filter2.hpp"



data_type filter2(data_type input) {
    static bool initialized = false;
    static data_type d1;
    static data_type d2;
    static data_type output;

    // One-time initialization
    if (!initialized) {
        d1 = 0.0;
        d2 = 0.0;
        output = 0.0;
        initialized = true;
    }


    // Scale the input
    accum_type scaled_input = input * scaleconst1;

    // Filter operations
    accum_type a2mul = d1 * a2;
    accum_type a3mul = d2 * a3;

    accum_type a2sum = scaled_input - a2mul;
    accum_type a1sum = a2sum - a3mul;

    output = a1sum - d2;

    // Update delay line (internal static variables)
    d2 = d1;
    d1 = a1sum;

    return output;
}
```

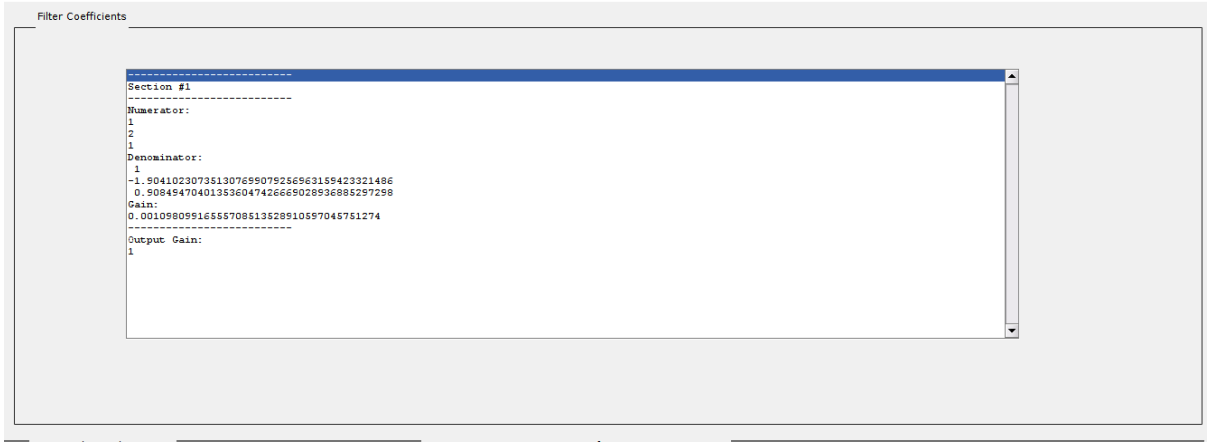
Synthesis Summary(solution1) filter2.cpp filter2.hpp

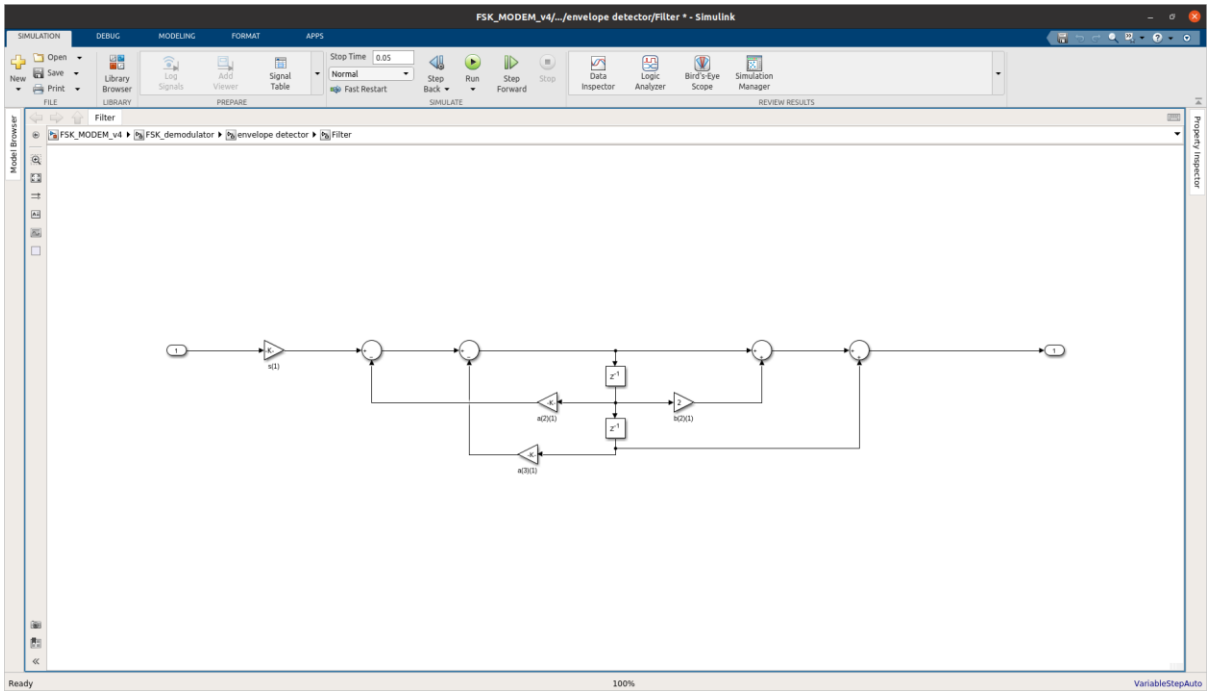
```cpp
#include <ap_int.h>
#include <ap_fixed.h>
#include <math.h>


// Data format
const int DataWordSize = 25;
const int DataIntSize = 3;
const float DataMaxVal = pow(2.0, DataIntSize-1);
typedef ap_fixed<DataWordSize, DataIntSize, AP_RND, AP_SAT, 0> data_type;


//coefficient constants and data types
const int CoeffWordSize = 18;
const int CoeffIntSize = 4;
typedef ap_fixed<CoeffWordSize, CoeffIntSize, AP_RND, AP_SAT, 0> coeff_type;

// Coefficients
const coeff_type scaleconst1 = 6.5174976764364075E-03;
const coeff_type a2 = -1.9830539690604794E+00;
const coeff_type a3 = 9.8696500464712711E-01;

typedef ap_fixed<DataWordSize+CoeffWordSize+3, DataIntSize+CoeffIntSize, AP_TRN, AP_WRAP, 0> accum_type;

// Processes one sample through the fixed-point IIR filter
data_type filter2(data_type input);
```



And the same story goes with the lowpass filter used in the envelope detector:


Block Parameters: Digital Filter Design2
File Edit Analysis Targets View Window Help
Current Filter Information
Structure: Direct-Form II, Second-Order Sections
Order: 2
Sections: 1
Stable: Yes
Source: Designed
Store Filter ...
Filter Manager ...
Filter Information
Design Specifications
Sample Rate : N/A (normalized frequency)
Response : Lowpass
Specification : N,F3dB
3-dB Point : 0.021
Filter Order : 2
Measurements
Sample Rate : N/A (normalized frequency)
Passband Edge : Unknown
3-dB Point : 0.021
6-dB Point : 0.028419
Stopband Edge : Unknown
Passband Ripple : Unknown
Stopband Atten. : Unknown
Transition Width : Unknown
Implementation Cost
Number of Multipliers : 4
Number of Adders : 4
Number of States : 2
Multiplications per Input Sample : 4
Additions per Input Sample : 4
Response Type
Lowpass
Highpass
Bandpass
Bandstop
Differentiator
Design Method
IIR
Butterworth
FIR
Equiripple
Filter Order
Specify order:
2
Minimum order
Options
There are no optional parameters for this design method.
Frequency Specifications
Units:
Hz
Fs:
1000000
Fc:
10800
Magnitude Specifications
The attenuation at cutoff frequencies is fixed at 3 dB (half the passband power)
Input processing:
Columns as channels (frame based)
Design Filter
Ready


Filter Coefficients
Section #1
Numerator:
1
2
1
Denominator:
1
-1.9041023073513076590792569631594233214 86
0.90849470401353604742666902893688529729 8
Gain:
0.0010980955165557085135289105970457512 74
Output Gain:
1

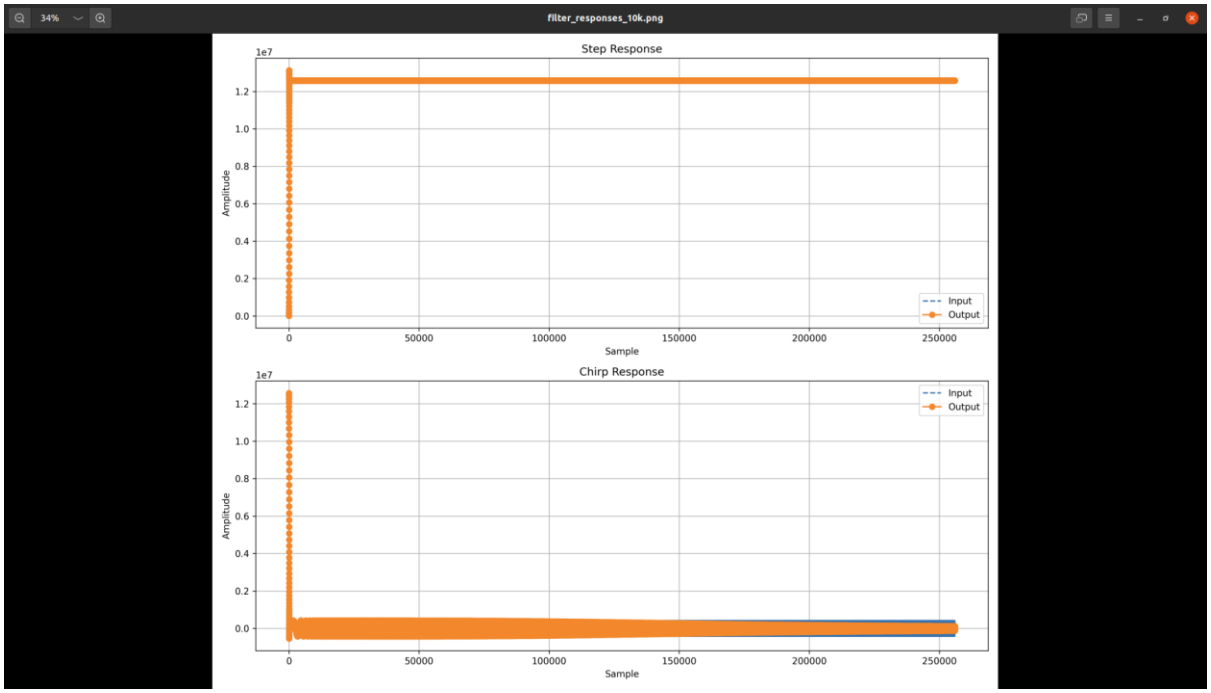


Synthesis Summary(solution1) filter_envelope_detector.cpp

```cpp
#include "filter_envelope_detector.hpp"

data_type applyIIRFilter(data_type filter_in) {
    static bool initialized = false;
    static data_type delay0;
    static data_type delay1;
    static data_type output_register;

    // One-time initialization
    if (!initialized) {
        delay0 = 0.0;
        delay1 = 0.0;
        output_register = 0.0;
        initialized = true;
    }

    // Signal path logic
    data_type input_register = filter_in;
    accum_type scale1 = input_register * scaleconst1;
    accum_type inputconv1 = scale1;

    accum_type a2mul1 = delay0 * coeff_a2_section1;
    accum_type a3mul1 = delay1 * coeff_a3_section1;
    accum_type b2mul1 = delay0 * coeff_b2_section1;

    accum_type a2sum1 = inputconv1 - a2mul1;
    accum_type a1sum1 = a2sum1 - a3mul1;
    accum_type b2sum1 = a1sum1 + b2mul1;
    accum_type b1sum1 = b2sum1 + delay1;

    output_register = b1sum1;

    // Update delay line
    delay1 = delay0;
    delay0 = a1sum1;

    //return output sample
    return output_register;

}
```

Synthesis Summary(solution1) | filter_envelope_detector.cpp | filter_envelope_detector_tb.cpp | filter_envelope_detector.hpp

```cpp
#include <ap_int.h>
#include <ap_fixed.h>
#include <math.h>


// Data format
const int DataWordSize = 25;
const int DataIntSize = 3;
const float DataMaxVal = pow(2.0, DataIntSize-1);
typedef ap_fixed<DataWordSize, DataIntSize, AP_RND, AP_SAT, 0> data_type;


//coefficient constants and data types
const int CoeffWordSize = 18;
const int CoeffIntSize = 4;
typedef ap_fixed<CoeffWordSize, CoeffIntSize, AP_RND, AP_SAT, 0> coeff_type;
const coeff_type scaleconst1 = 1.0980991655570851E-03;
const coeff_type coeff_b1_section1 = 1.0000000000000000E+00;
const coeff_type coeff_b2_section1 = 2.0000000000000000E+00;
const coeff_type coeff_b3_section1 = 1.0000000000000000E+00;
const coeff_type coeff_a2_section1 = -1.9041023073513077E+00;
const coeff_type coeff_a3_section1 = 9.0849470401353605E-01;


typedef ap_fixed<DataWordSize+CoeffWordSize+3, DataIntSize+CoeffIntSize, AP_TRN, AP_WRAP, 0> accum_type;

```



As we can see, by using HLS, all of the three filters needed to build the FSK non coherent modem have been built.

THE VIVADO PROJECT

The Vivado design is managed via bash and tcl scripts:

vivado_build.sh
~/Downloads/TUTORIAL_FSK/FSK_MODEM/VIVADO

```
#!/bin/bash

# Exit on any error
set -e

echo "Starting Vivado build process..."
echo "Running: prj.tcl"
vivado -mode batch -source prj.tcl

echo "Vivado build completed successfully."

```

sh Tab Width: 8 Ln 1, Col 1 INS

```

#add simulation sources
set_property SOURCE_SET sources_1 [get_filesets sim_1]
add_files -fileset sim_1 -norecurse ./tb_modem.vhd

#add design sources
add_files -norecurse {./clocked_comparator_25bit.vhd ./mux2to1_32bit.vhd ./sinewave_generator.vhd}
update_compile_order -fileset sources_1

#build block design
source ./bd.tcl

#create hdl wrapper
make_wrapper -files [get_files ./project_1/project_1.srcs/sources_1/bd/design_1/design_1.bd] -top
add_files -norecurse ./project_1/project_1.gen/sources_1/bd/design_1/hdl/design_1_wrapper.vhd
update_compile_order -fileset sources_1

#add waveform configuration file
add_files -fileset sim_1 -norecurse ./tb_design_1_wrapper_behav.wcfg
set_property xsim.view ./tb_design_1_wrapper_behav.wcfg [get_filesets sim_1]

```

Tcl Tab Width: 8 Ln 53, Col 2 INS

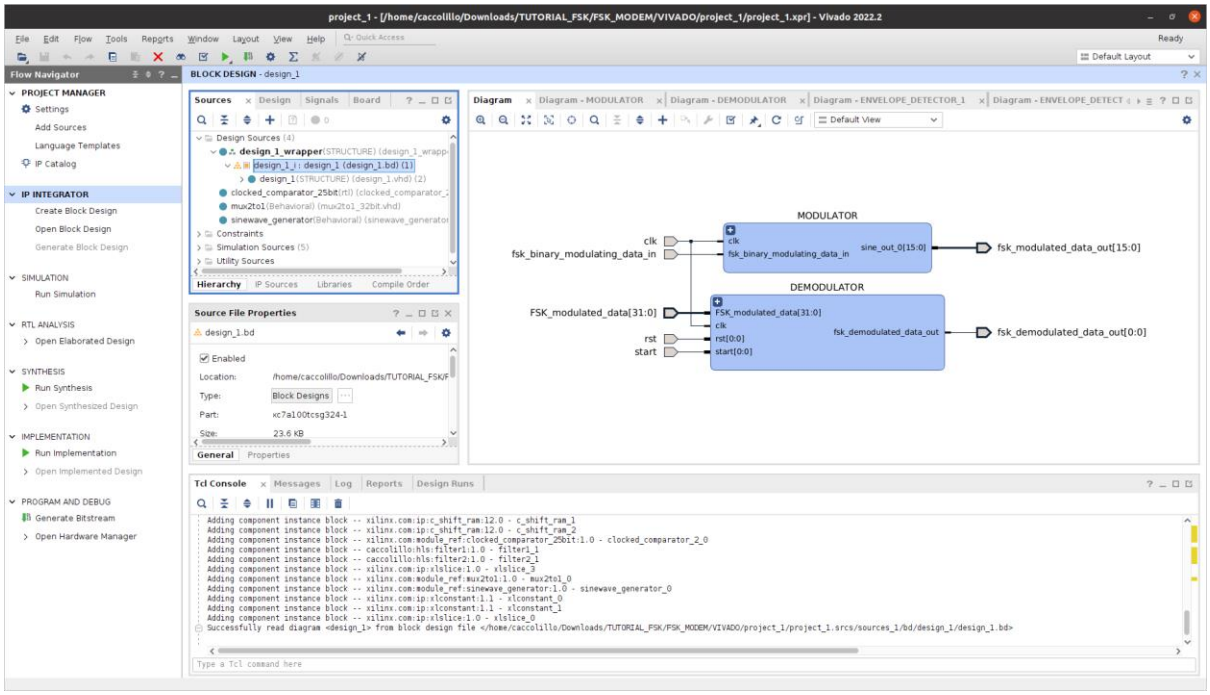
prj.tcl
~/Downloads/TUTORIAL_FSK/FSK_MODEM/VIVADO

vivado_build.sh | prj.tcl

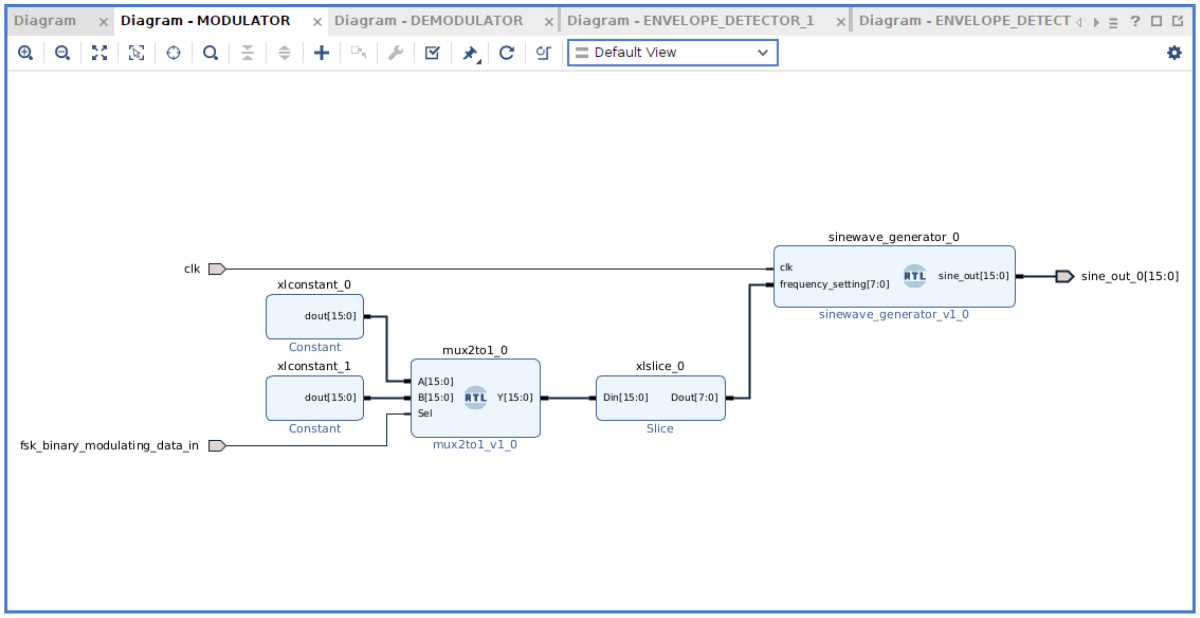
```
# Set the reference directory for source file relative paths (by default the value is script directory path)
set origin_dir "."


# Set the project name
set _xil_proj_name_ "project_1"

# Set the directory path for the original project from where this script was exported
set orig_proj_dir "[file normalize "$origin_dir/project_1"]"


# Create project
create_project ${_xil_proj_name_} ./${_xil_proj_name_} -part xc7a100tcsg324-1

# Set the directory path for the new project
set proj_dir [get_property directory [current_project]]

# Set project properties
set obj [current_project]
set_property -name "board_part" -value "digilentinc.com:arty-a7-100:part0:1.1" -objects $obj
set_property -name "default_lib" -value "xil_defaultlib" -objects $obj
set_property -name "enable_resource_estimation" -value "0" -objects $obj
set_property -name "enable_vhdl_2008" -value "1" -objects $obj
set_property -name "ip_cache_permissions" -value "read write" -objects $obj
set_property -name "ip_output_repo" -value "$proj_dir/${_xil_proj_name_}.cache/ip" -objects $obj
set_property -name "mem.enable_memory_map_generation" -value "1" -objects $obj
set_property -name "platform.board_id" -value "arty-a7-100" -objects $obj
set_property -name "revised_directory_structure" -value "1" -objects $obj
set_property -name "sim.central_dir" -value "$proj_dir/${_xil_proj_name_}.ip_user_files" -objects $obj
set_property -name "sim.ip.auto_export_scripts" -value "1" -objects $obj
set_property -name "simulator_language" -value "Mixed" -objects $obj
set_property -name "sim_compile_state" -value "1" -objects $obj
set_property -name "target_language" -value "VHDL" -objects $obj
set_property -name "webtalk.activehdl_export_sim" -value "52" -objects $obj
set_property -name "webtalk.modelsim_export_sim" -value "52" -objects $obj
set_property -name "webtalk.questa_export_sim" -value "52" -objects $obj
set_property -name "webtalk.riviera_export_sim" -value "52" -objects $obj
set_property -name "webtalk.vcs_export_sim" -value "52" -objects $obj
set_property -name "webtalk.xcelium_export_sim" -value "2" -objects $obj
set_property -name "webtalk.xsim_export_sim" -value "52" -objects $obj
set_property -name "webtalk.xsim_launch_sim" -value "142" -objects $obj


# Set IP repository paths
set obj [get_filesets sources_1]
if { $obj != {} } {
   set_property "ip_repo_paths" "[file normalize "$origin_dir/../HLS"]" $obj

   # Rebuild user ip_repo's index before adding any source files
   update_ip_catalog -rebuild
}
```

Tcl Tab Width: 8 Ln 74, Col 25 INS

By launching the bash script, the Vivado project gets built:



The modem has been implemented in a graphical fashion in a block design:



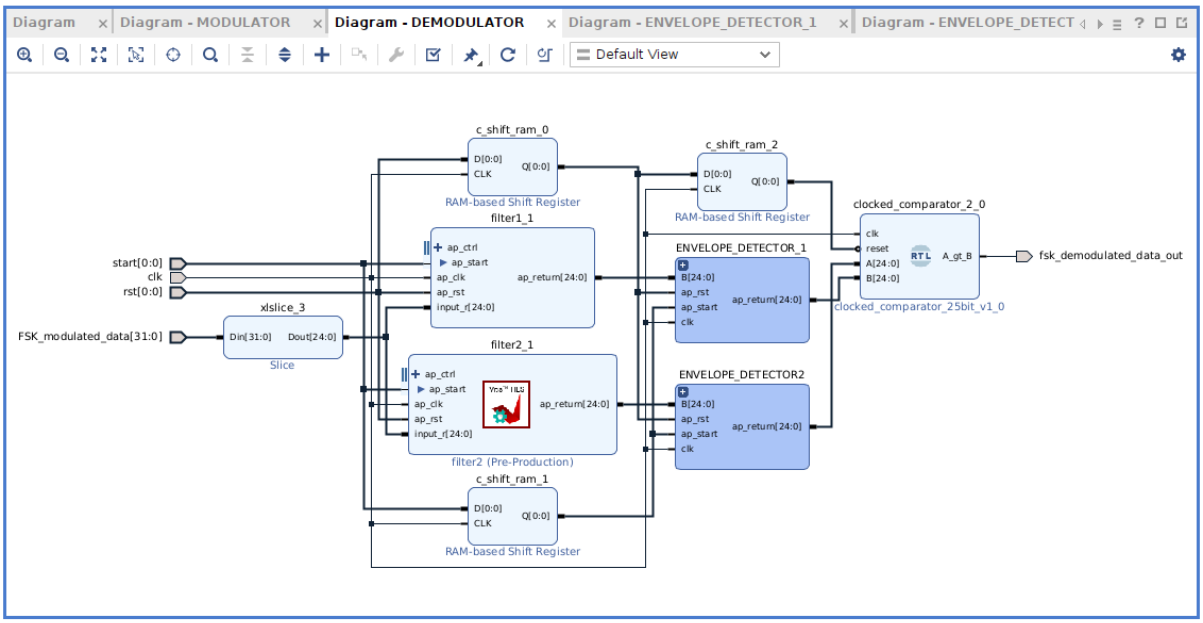


Diagram
Diagram - MODULATOR
Diagram - DEMODULATOR
Diagram - ENVELOPE_DETECTOR_1
Diagram - ENVELOPE_DETECT
Default View
c_shift_ram_0
D[0:0]
CLK
Q[0:0]
RAM-based Shift Register
c_shift_ram_2
RAM-based Shift Register
clocked_comparator_2_0
clk
reset
A[24:0]
B[24:0]
RTL
A_gt_B
fsk_demodulated_data_out
clocked_comparator_25bit_v1_0
filter1_1
ap_ctrl
ap_start
ap_clk
ap_rst
input_r[24:0]
ap_return[24:0]
ENVELOPE_DETECTOR_1
B[24:0]
start[0:0]
clk
rst[0:0]
xlslice_3
FSK_modulated_data[31:0]
Din[31:0]
Dout[24:0]
Slice
filter2_1
Vitis™ HLS
filter2 (Pre-Production)
ENVELOPE_DETECTOR2
c_shift_ram_1

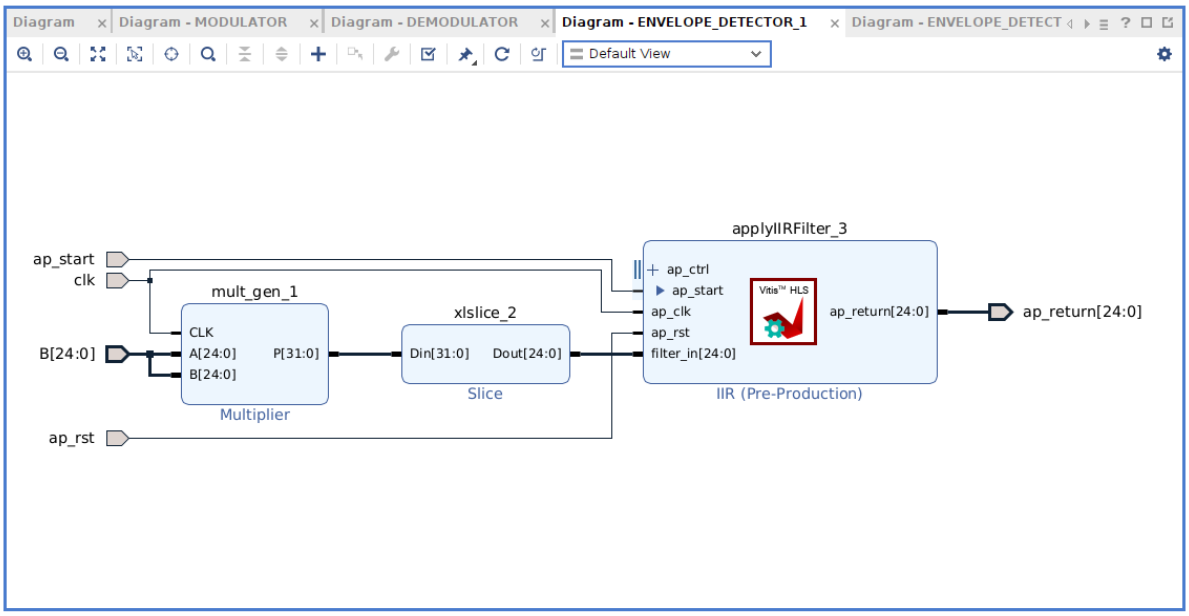
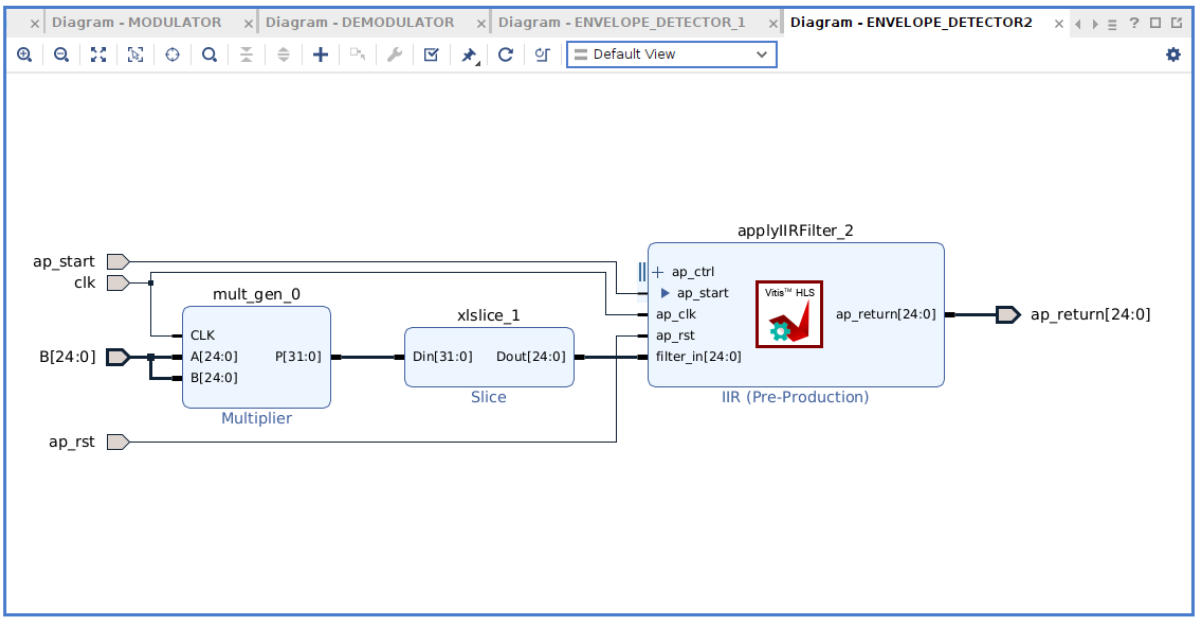
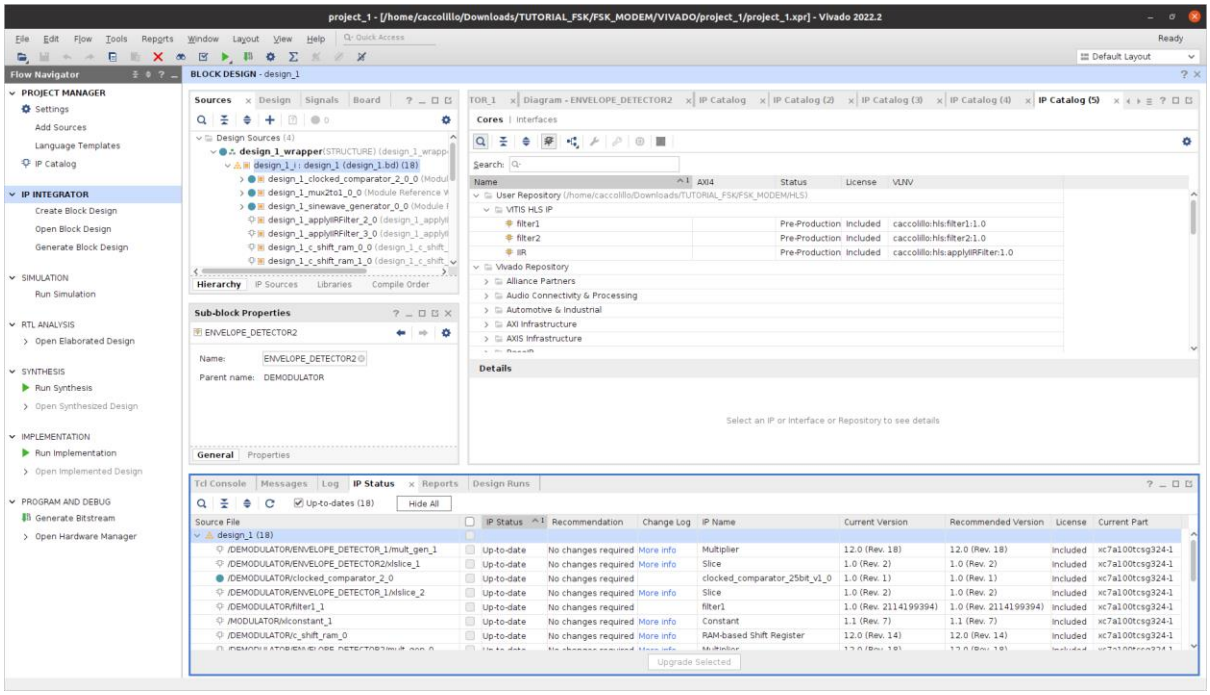


Diagram
Diagram - MODULATOR
Diagram - DEMODULATOR
Diagram - ENVELOPE_DETECTOR_1
Diagram - ENVELOPE_DETECT
Default View
applyIIRFilter_3
ap_start
clk
mult_gen_1
CLK
A[24:0]
B[24:0]
P[31:0]
Multiplier
xlslice_2
Din[31:0]
Dout[24:0]
Slice
B[24:0]
ap_rst
ap_ctrl
ap_clk
filter_in[24:0]
Vitis™ HLS
ap_return[24:0]
IIR (Pre-Production)



As we can see, the three HLS filters appear under IP-catalogue:



In addition to that, three VHDL files have been prepared:

- A clocked comparator:

clocked_comparator_25bit.vhd

/home/caccolillo/Downloads/TUTORIAL_FSK/FSK_MODEM/VIVADO/clocked_comparator_25bit.vhd

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

entity clocked_comparator_25bit is
    port (
        clk      : in  std_logic;
        reset    : in  std_logic; -- optional reset
        A        : in  std_logic_vector(24 downto 0);
        B        : in  std_logic_vector(24 downto 0);
        A_gt_B   : out std_logic
    );
end entity;

architecture rtl of clocked_comparator_25bit is
    signal A_signed, B_signed : signed(24 downto 0);
    signal result             : std_logic := '0';
begin

    -- Convert inputs to signed
    A_signed <= signed(A);
    B_signed <= signed(B);

    -- Clocked process
    process(clk, reset)
    begin
        if reset = '1' then
            result <= '0';
        elsif rising_edge(clk) then
            if A_signed > B_signed then
                result <= '1';
            else
                result <= '0';
            end if;
        end if;
    end process;

    -- Output assignment
    A_gt_B <= result;

end architecture;
```

- A MUX 2:1:

mux2to1_32bit.vhd

/home/caccolillo/Downloads/TUTORIAL_FSK/FSK_MODEM/VIVADO/mux2to1_32bit.vhd

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux2to1 is
    Port (
        A     : in  std_logic_vector(15 downto 0);  -- Input A
        B     : in  std_logic_vector(15 downto 0);  -- Input B
        Sel   : in  std_logic;                      -- Select line
        Y     : out std_logic_vector(15 downto 0)   -- Output
    );
end mux2to1;

architecture Behavioral of mux2to1 is
begin
    process(A, B, Sel)
    begin
        if Sel = '0' then
            Y <= A;
        else
            Y <= B;
        end if;
    end process;
end Behavioral;
```

- A sinewave generator:

sinewave_generator.vhd

/home/caccolillo/Downloads/TUTORIAL_FSK/FSK_MODEM/VIVADO/sinewave_generator.vhd

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity sinewave_generator is
    Port (
        clk              : in  std_logic;
        frequency_setting : in  std_logic_vector(7 downto 0); -- 8-bit frequency control
        sine_out         : out std_logic_vector(15 downto 0) -- 16-bit sine wave output
    );
end sinewave_generator;

architecture Behavioral of sinewave_generator is

    -- Constants
    constant LUT_SIZE : integer := 256;




type sine_table_type is array(0 to 255) of unsigned(15 downto 0);

signal sine_table : sine_table_type := (
      0 => to_unsigned(35446, 16),
      1 => to_unsigned(36182, 16),
      2 => to_unsigned(36918, 16),
      3 => to_unsigned(37652, 16),
      4 => to_unsigned(38385, 16),
      5 => to_unsigned(39118, 16),
      6 => to_unsigned(39847, 16),
      7 => to_unsigned(40574, 16),
      8 => to_unsigned(41298, 16),
      9 => to_unsigned(42019, 16),
     10 => to_unsigned(42735, 16),
     11 => to_unsigned(43447, 16),
     12 => to_unsigned(44154, 16),
     13 => to_unsigned(44855, 16),
     14 => to_unsigned(45551, 16),
     15 => to_unsigned(46241, 16),
     16 => to_unsigned(46924, 16),
     17 => to_unsigned(47602, 16),
     18 => to_unsigned(48271, 16),
     19 => to_unsigned(48933, 16),
     20 => to_unsigned(49586, 16),
     21 => to_unsigned(50232, 16),
     22 => to_unsigned(50868, 16),
     23 => to_unsigned(51494, 16),
     24 => to_unsigned(52111, 16),
     25 => to_unsigned(52718, 16),
     26 => to_unsigned(53314, 16),
     27 => to_unsigned(53899, 16),
     28 => to_unsigned(54473, 16),
     29 => to_unsigned(55035, 16),
     30 => to_unsigned(55586, 16),
     31 => to_unsigned(56123, 16),
     32 => to_unsigned(56658, 16),
     33 => to_unsigned(57178, 16),
     34 => to_unsigned(57687, 16),
     35 => to_unsigned(58182, 16),
     36 => to_unsigned(58663, 16),
     37 => to_unsigned(59130, 16),
     38 => to_unsigned(59583, 16),
```

sinewave_generator.vhd

/home/caccolillo/Downloads/TUTORIAL_FSK/FSK_MODEM/VIVADO/sinewave_generator.vhd

```
    227 => to_unsigned(12170, 16),
    228 => to_unsigned(12878, 16),
    229 => to_unsigned(13604, 16),
    230 => to_unsigned(14349, 16),
    231 => to_unsigned(15113, 16),
    232 => to_unsigned(15895, 16),
    233 => to_unsigned(16695, 16),
    234 => to_unsigned(17513, 16),
    235 => to_unsigned(18349, 16),
    236 => to_unsigned(19203, 16),
    237 => to_unsigned(20075, 16),
    238 => to_unsigned(20964, 16),
    239 => to_unsigned(21870, 16),
    240 => to_unsigned(22794, 16),
    241 => to_unsigned(23734, 16),
    242 => to_unsigned(24692, 16),
    243 => to_unsigned(25666, 16),
    244 => to_unsigned(26658, 16),
    245 => to_unsigned(27666, 16),
    246 => to_unsigned(28689, 16),
    247 => to_unsigned(29730, 16),
    248 => to_unsigned(30787, 16),
    249 => to_unsigned(31860, 16),
    250 => to_unsigned(32949, 16),
    251 => to_unsigned(34054, 16),
    252 => to_unsigned(35174, 16),
    253 => to_unsigned(36308, 16),
    254 => to_unsigned(37459, 16),
    255 => to_unsigned(38615, 16)
);


    -- Internal signals
    signal phase_accumulator : unsigned(15 downto 0) := (others => '0');
    signal phase_increment   : unsigned(15 downto 0);

begin

    -- Calculate phase increment from frequency setting
    process(frequency_setting)
    begin
        -- Map 8-bit frequency setting to 16-bit phase increment
        -- Simple linear mapping: frequency_setting * constant
        phase_increment <= resize(unsigned(frequency_setting) * 4, 16);  -- Scale factor is arbitrary
    end process;

    -- Phase accumulator and LUT access
    process(clk)
    begin
        if rising_edge(clk) then
            phase_accumulator <= phase_accumulator + phase_increment;
        end if;
    end process;

    -- Output sine wave sample from LUT
    sine_out <= std_logic_vector(sine_table(to_integer(phase_accumulator(15 downto 8))));

end Behavioral;
```

In order to quickly evaluate the modem, a simple VHDL testbench has been prepared:

tb_modem.vhd

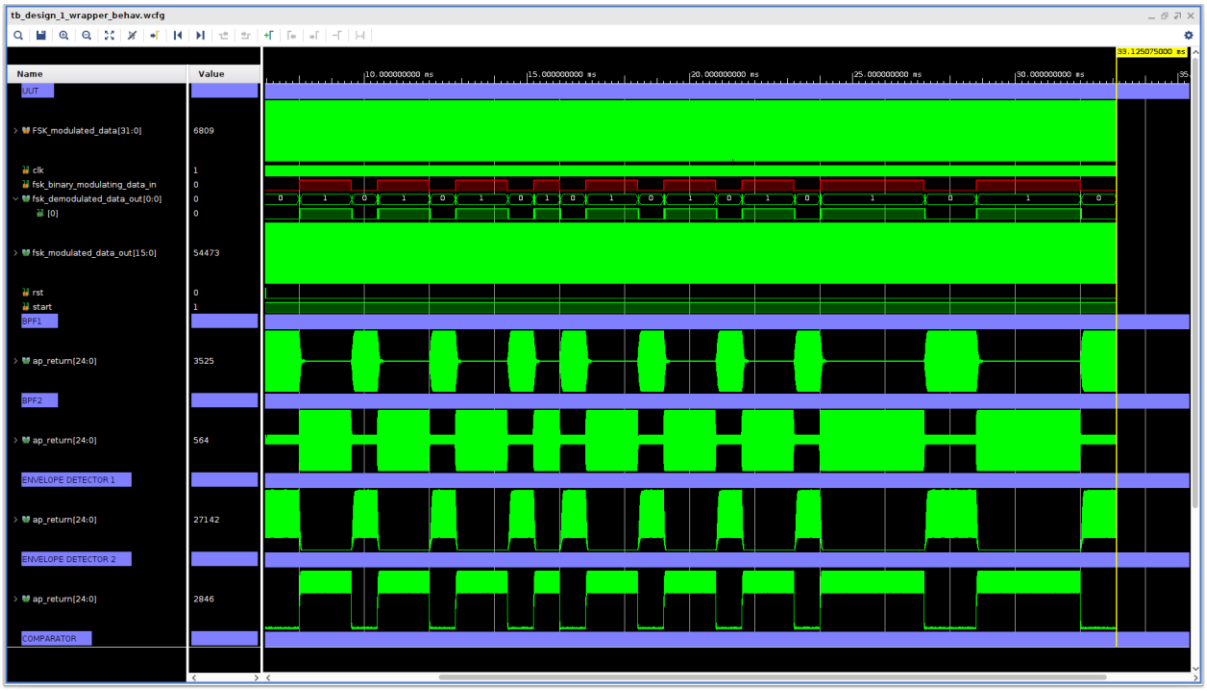
/home/caccolillo/Downloads/TUTORIAL_FSK/FSK_MODEM/VHDADO/tb_modem.vhd

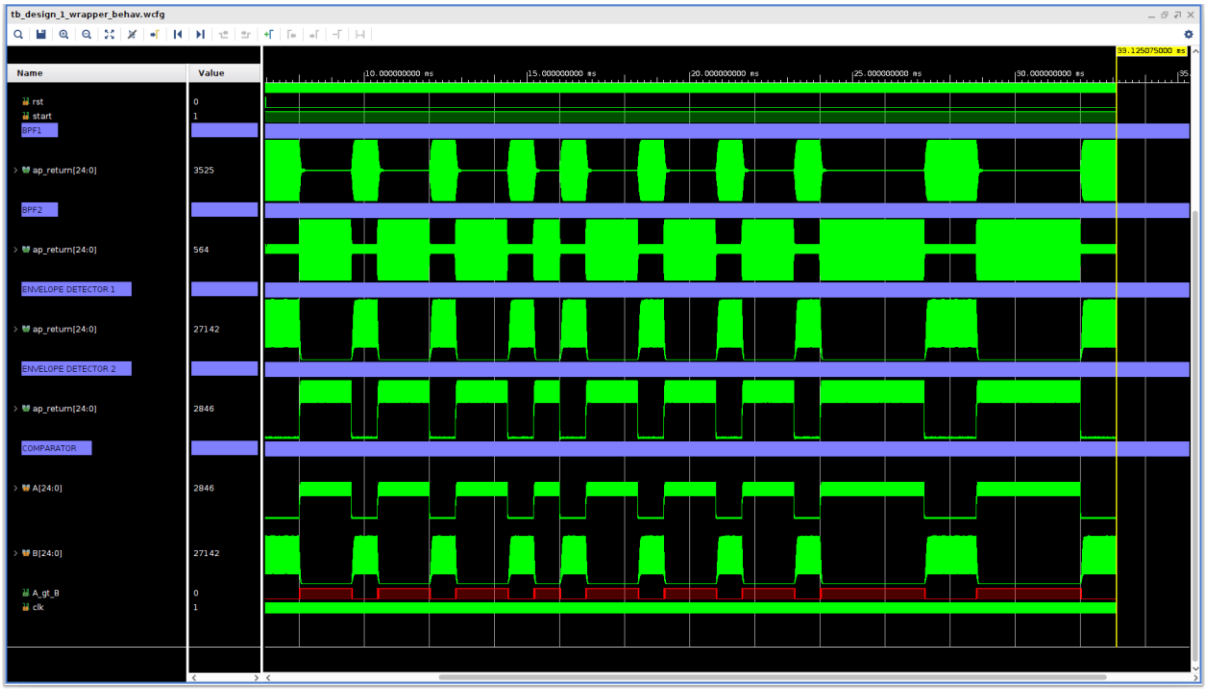
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.numeric_std.all;

entity tb_design_1_wrapper is
end tb_design_1_wrapper;

architecture sim of tb_design_1_wrapper is

  -- Component Declaration
  component design_1_wrapper
    port (
      FSK_modulated_data : in STD_LOGIC_VECTOR ( 31 downto 0 );
      clk : in STD_LOGIC;
      fsk_binary_modulating_data_in : in STD_LOGIC;
      fsk_demodulated_data_out : out STD_LOGIC_VECTOR ( 0 to 0 );
      fsk_modulated_data_out : out STD_LOGIC_VECTOR ( 15 downto 0 );
      rst : in STD_LOGIC;
      start : in STD_LOGIC
    );
  end component;

  -- Signals
  signal en                         : std_logic := '0';
  signal clk                        : std_logic := '0';
  signal rst                        : std_logic := '1';
  signal start                      : std_logic := '1';
  signal FSK_modulated_data         : std_logic_vector(31 downto 0) := (others => '0');
  signal fsk_demodulated_data_out   : std_logic_vector(0 to 0);
  signal lfsr_reg : std_logic_vector(31 downto 0) := (others => '1');  -- Non-zero seed
  signal fsk_modulated_data_out : STD_LOGIC_VECTOR ( 15 downto 0 );

  -- Constants
  constant CLK_PERIOD       : time := 10 ns; -- 100 MHz

begin

  -- DUT Instantiation
  uut: design_1_wrapper
    port map (
      FSK_modulated_data          => FSK_modulated_data,
      fsk_demodulated_data_out    => fsk_demodulated_data_out,
      clk                         => clk,
      rst                         => rst,
      start                       => start,
      fsk_binary_modulating_data_in => lfsr_reg(0),
      fsk_modulated_data_out      => fsk_modulated_data_out
    );

--FSK_modulated_data <= std_logic_vector(resize(signed(fsk_modulated_data_out), 32));
FSK_modulated_data <= std_logic_vector(resize(unsigned(fsk_modulated_data_out(15 downto 3)), 32));

  -- Clock generation (100 MHz)
  clk_process : process
  begin
    while true loop
      clk <= '0';
      wait for CLK_PERIOD / 2;
      clk <= '1';
      wait for CLK_PERIOD / 2;
    end loop;
```

```
  end process;

  en_proc:process
  begin
    for i in 0 to 80000 loop
      wait until rising_edge(clk);
    end loop;
    en <= '1';
    wait until rising_edge(clk);
    en <= '0';
  end process;

  -- Stimulus process
  stim_proc : process
    variable cycle_count       : integer := 0;
    variable start_cycle_count : integer := 0;
    variable start_count       : integer := 0;
    variable square_state      : boolean := false;
  begin
    wait for CLK_PERIOD * 1; -- allow some initial delay

    -- Reset active for first 50 clock cycles
    wait for CLK_PERIOD * 50;
    rst <= '0';

    wait for CLK_PERIOD;

    wait; -- End simulation
  end process;


    process(clk, rst)
    begin
        if rst = '1' then
            lfsr_reg <= x"00001000";  -- Reset to non-zero value
        elsif rising_edge(clk) then
          if(en = '1')then
                -- Polynomial: x^32 + x^22 + x^2 + x^1 + 1 (tap positions: 32,22,2,1)
                -- Feedback is XOR of taps
            lfsr_reg <= lfsr_reg(30 downto 0) &
                        (lfsr_reg(31) xor lfsr_reg(21) xor lfsr_reg(1) xor lfsr_reg(0));
          end if;
        end if;
    end process;

end sim;
```

A pseudo random sequence generator feeds the binary input of the modulator.

The output of the modulator is connected to the input of the demodulator, and its binary output gets observed in the simulation window:



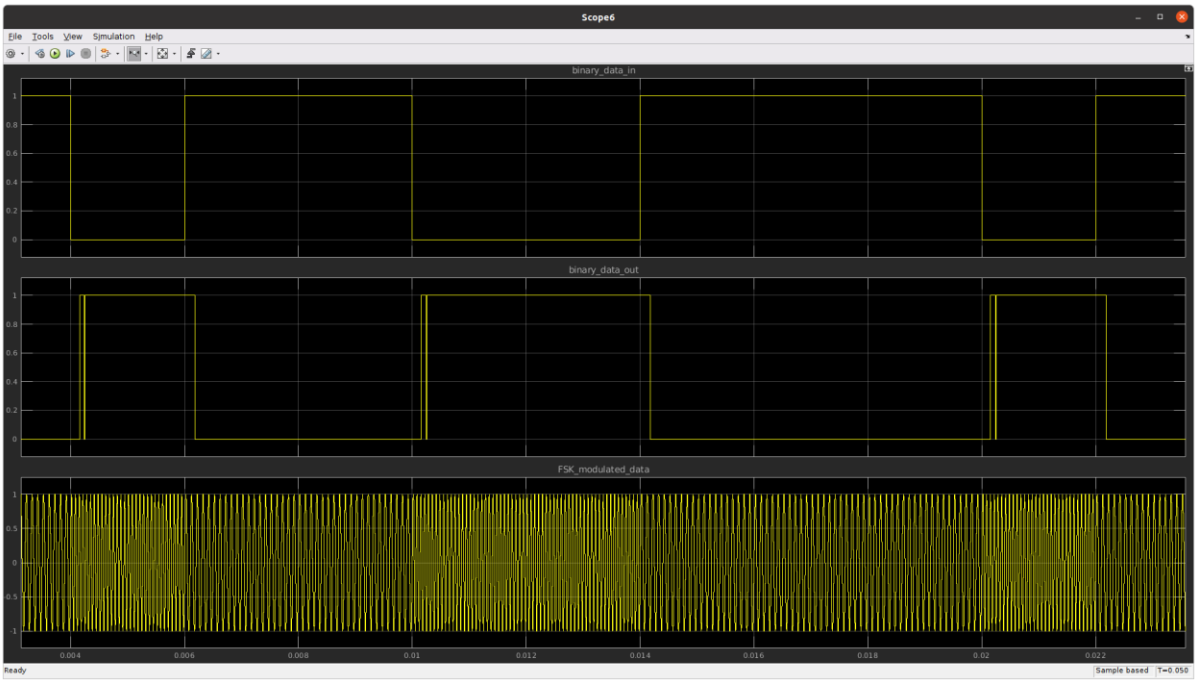


By zooming in across the binary input transitions:

We can spot glitches on the demodulated data.

This is due to the filters not being too much aggressive, in favour of a reduced HW complexity.

The FPGA implementation matches the behaviour of the simulink model:



This usually should not be an issue, as the transmission is serial, and we could think of inserting an UART to serialise and de-serialise the parallel data to be transmitted via the FSK modem.

THE VIVADO UART TESTBENCH

UART combined with FSK modulation is used in systems that require simple, reliable data communication over noisy or long-distance channels.

This solution, combines the simplicity of UART with the noise resistance and transmission range of FSK.

Some common applications are:

- Embedded and Industrial Systems: Wireless or long-distance communication using UART interfaced with an FSK modem.
- Wireless Sensor Networks: Microcontrollers send UART data to FSK transceivers for low-power RF communication.
- Audio-Based Data Transfer: UART data is converted to FSK signals for transmission over telephone lines or audio channels.
- Amateur Radio: Used in digital radio protocols like AX.25 and APRS for serial-to-radio communication.
- Smart Metering: Utility meters transmit data using UART-to-FSK over RF or power lines.
- Automotive and Rail Systems: UART data modulated via FSK for robust transmission over noisy wired channels.
- Satellite Communication: Small satellites use UART with FSK radios for low-power ground station links.

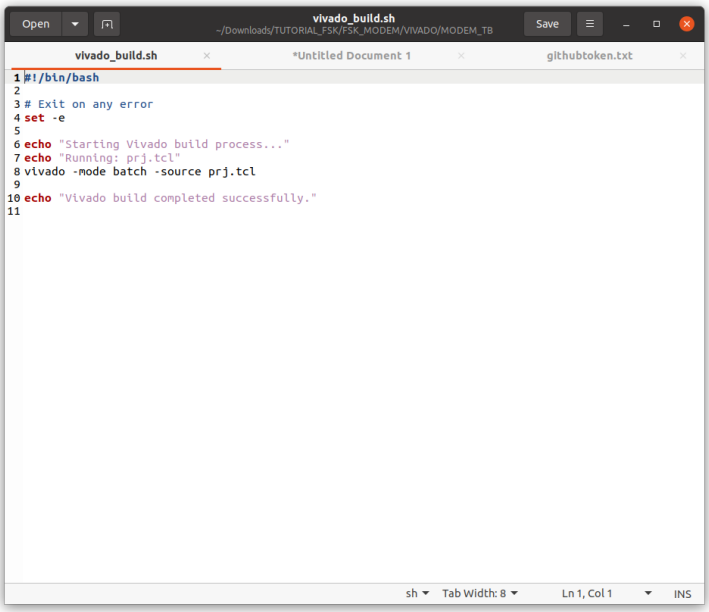
Here’s some examples of commercial products leveraging this technical solution:

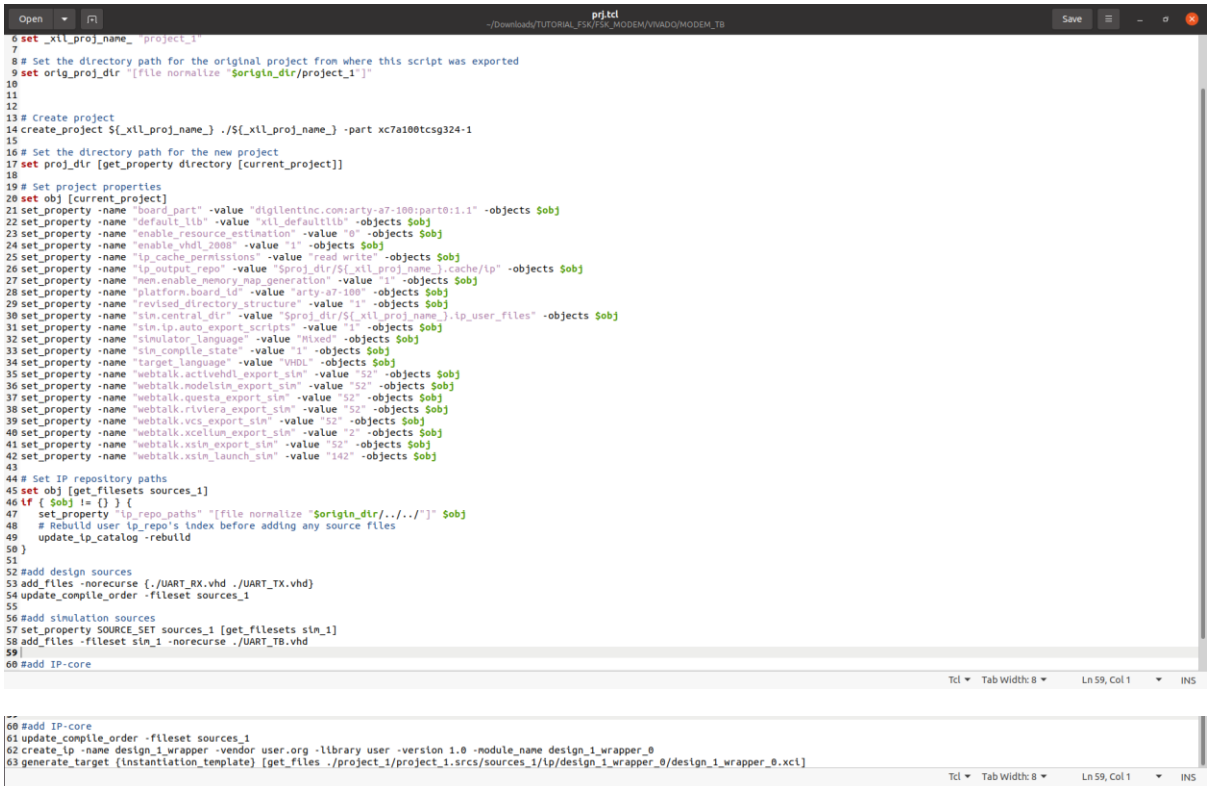
- rf & wireless modules like rfm69 and sx127x use uart or spi interfaces and modulate data using fsk, gfsk, or ook. they're common in iot and remote sensing.
- audio fsk modems such as the tcm3105 and mobilinkd tnc3 handle legacy protocols like bell 202 and afsk1200. they rely on uart for input and are used in scada systems and amateur radio.

- smart metering chips like st7580 and tms320c2000 paired with afe031 apply fsk, psk, or ofdm modulation. uart links digital control to analog modulation hardware for plc communication.
- satellite systems such as amsat's fox-1 use uart to transfer telemetry data via ax.25 protocol into fsk modems, typically at 1200 bps.
- industrial scada modems in the rad series bridge uart or rs-232 interfaces to modulated fsk links, enabling robust data transmission in legacy infrastructure.

In order to test the behaviour of the FSK modem designed so far, when coupled with an UART, a Vivado separate project has been created.

It is managed via bash and tcl scripts:





A VHDL testbench has been created, in order to simulate the modem link:

UART_TB.vhd

/home/caccolillo/Downloads/TUTORIAL_FSK/FSK_MODEM/VIVADO/MODEM_TB/UART_TB.vhd

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity uart_tb is
6  end uart_tb;
7
8  architecture Behavioral of uart_tb is
9      constant CLK_PERIOD : time := 10 ns;  -- 100 MHz
10
11     signal clk       : std_logic := '0';
12     signal tx_start  : std_logic := '0';
13     signal tx_data   : std_logic_vector(7 downto 0) := x"00"; -- test pattern
14     signal tx_line   : std_logic;
15     signal rx_data   : std_logic_vector(7 downto 0);
16     signal rx_valid  : std_logic;
17     signal busy      : std_logic;
18     signal tx_start_0 : std_logic;
19     signal fsk_demodulated_data_out : std_logic_vector(0 DOWNTO 0);
20     signal start      : std_logic:='0';
21
22
23   component design_1_wrapper is
24     port (
25       clk : in STD_LOGIC;
26       rst : in STD_LOGIC;
27       start : in STD_LOGIC;
28       FSK_modulated_data : in STD_LOGIC_VECTOR ( 31 downto 0 );
29       fsk_modulated_data_out : out STD_LOGIC_VECTOR ( 15 downto 0 );
30       tx_start_0 : in STD_LOGIC;
31       busy_0 : out STD_LOGIC;
32       data_in_0 : in STD_LOGIC_VECTOR ( 7 downto 0 );
33       rx_valid_0 : out STD_LOGIC;
34       rx_data_0 : out STD_LOGIC_VECTOR ( 7 downto 0 )
35     );
36   end component;
37
38   component design_1_wrapper_0
39     PORT (
40       FSK_modulated_data : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
41       clk : IN STD_LOGIC;
42       fsk_binary_modulating_data_in : IN STD_LOGIC;
43       fsk_demodulated_data_out : OUT STD_LOGIC_VECTOR(0 DOWNTO 0);
44       fsk_modulated_data_out : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
45       rst : IN STD_LOGIC;
46       start : IN STD_LOGIC
47     );
48   end component;
49
50   signal fsk_modulated_data_out : std_logic_vector ( 15 downto 0 );
51   signal FSK_modulated_data     : std_logic_vector(31 downto 0) := (others => '0');
52   signal rst                    : std_logic := '1';
53
54
55
56 begin
57
58   FSK_modulated_data <= std_logic_vector(resize(unsigned(fsk_modulated_data_out(15 downto 3)), 32));
59
60
61
62   design_1_wrapper_inst : design_1_wrapper_0
```

UART_TB.vhd

/home/caccolillo/Downloads/TUTORIAL_FSK/FSK_MODEM/VIVADO/MODEM_TB/UART_TB.vhd

```
61
62      design_1_wrapper_inst : design_1_wrapper_0
63      PORT MAP (
64        FSK_modulated_data => FSK_modulated_data,
65        clk => clk,
66        fsk_binary_modulating_data_in => tx_line,
67        fsk_demodulated_data_out => fsk_demodulated_data_out,
68        fsk_modulated_data_out => fsk_modulated_data_out,
69        rst => rst,
70        start => start
71      );
72
73      -- Clock generation
74      clk_process : process
75      begin
76        while true loop
77          clk <= '0';
78          wait for CLK_PERIOD / 2;
79          clk <= '1';
80          wait for CLK_PERIOD / 2;
81        end loop;
82      end process;
83
84      -- DUT Instantiation (TX)
85      uart_tx_inst : entity work.uart_tx
86      port map (
87        clk      => clk,
88        tx_start => tx_start,
89        data_in  => tx_data,
90        tx_line  => tx_line,
91        busy     => busy
92      );
93
94      -- DUT Instantiation (RX)
95      uart_rx_inst : entity work.uart_rx
96        port map (
97          clk      => clk,
98          rx_line  => fsk_demodulated_data_out(0),
99          rx_data  => rx_data,
100         rx_valid => rx_valid
101       );
102
103     -- Stimulus
104     stimulus : process
105     begin
106
107       wait for CLK_PERIOD * 1; -- allow some initial delay
108
109       -- Reset active for first 50 clock cycles
110       wait for CLK_PERIOD * 50;
111       rst <= '0';
112
113       wait for CLK_PERIOD;
114
115
116       wait for 100 ns;
117       tx_start <= '1';
118       wait for CLK_PERIOD;
119       tx_start <= '0';
120       wait for CLK_PERIOD;
121       start <= '1';
122
```

```
    start <= '1';


    for i in 0 to 1024 loop
      wait until rx_valid='1';
      wait for 230 us;

      -- Assert that tx_data = rx_data + 1
      if not (tx_data = x"00" and rx_data = x"00") then
        assert unsigned(tx_data) = (unsigned(rx_data) + 1) mod 256
        report "Mismatch: tx_data /= rx_data + 1 (mod 256)"
        severity failure;
      end if;

      tx_start <= '1';
      wait for CLK_PERIOD;
      tx_start <= '0';
      wait for CLK_PERIOD;
      tx_data <= std_logic_vector(unsigned(tx_data) + 1);

    end loop;

    report "End of simulation..."
        severity failure;

  end process;

end Behavioral;
```

No noise is present though.

A simple UART TX and rx vhdl files have been added to the design files:

UART_RX.vhd

/home/caccolillo/Downloads/TUTORIAL_FSK/FSK_MODEM/VIVADO/MODEM_TB/UART_RX.vhd

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity uart_rx is
6      Port (
7          clk       : in  std_logic;
8          rx_line   : in  std_logic;
9          rx_data   : out std_logic_vector(7 downto 0);
10         rx_valid  : out std_logic
11     );
12 end uart_rx;
13
14 architecture Behavioral of uart_rx is
15     constant CLK_FREQ    : integer := 100_000_000; -- Hz
16     constant BAUD_RATE   : integer := 9600;
17     constant BIT_PERIOD  : integer := CLK_FREQ / BAUD_RATE;
18     constant SAMPLE_POINT: integer := BIT_PERIOD / 2;
19
20     type rx_state_type is (IDLE, START_BIT, DATA_BITS, STOP_BIT);
21     signal rx_state    : rx_state_type := IDLE;
22     signal bit_index   : integer range 0 to 7 := 0;
23     signal rx_shift    : std_logic_vector(7 downto 0) := (others => '0');
24     signal rx_counter  : integer := 0;
25     signal rx_valid_reg: std_logic := '0';
26 begin
27
28     process(clk)
29     begin
30         if rising_edge(clk) then
31             rx_valid_reg <= '0';
32             case rx_state is
33                 when IDLE =>
34                     if rx_line = '0' then  -- detect start bit
35                         rx_state <= START_BIT;
36                         rx_counter <= 0;
37                     end if;
38
39                 when START_BIT =>
40                     rx_counter <= rx_counter + 1;
41                     if rx_counter = SAMPLE_POINT then
42                         rx_counter <= 0;
43                         rx_state <= DATA_BITS;
44                         bit_index <= 0;
45                     end if;
46
47                 when DATA_BITS =>
48                     rx_counter <= rx_counter + 1;
49                     if rx_counter = BIT_PERIOD then
50                         rx_counter <= 0;
51                         rx_shift(bit_index) <= rx_line;
52                         bit_index <= bit_index + 1;
53                         if bit_index = 7 then
54                             rx_state <= STOP_BIT;
55                         end if;
56                     end if;
57
58                 when STOP_BIT =>
59                     rx_counter <= rx_counter + 1;
60                     if rx_counter = BIT_PERIOD then
61                         rx_counter <= 0;
62                         rx_data <= rx_shift;
63                         rx_valid_reg <= '1';
64                         rx_state <= IDLE;
65                     end if;
66             end case;
67         end if;
68     end process;
69
70     rx_valid <= rx_valid_reg;
71
72 end Behavioral;
73
```

UART_TX.vhd

/home/caccolillo/Downloads/TUTORIAL_FSK/FSK_MODEM/VIVADO/MODEM_TB/UART_TX.vhd

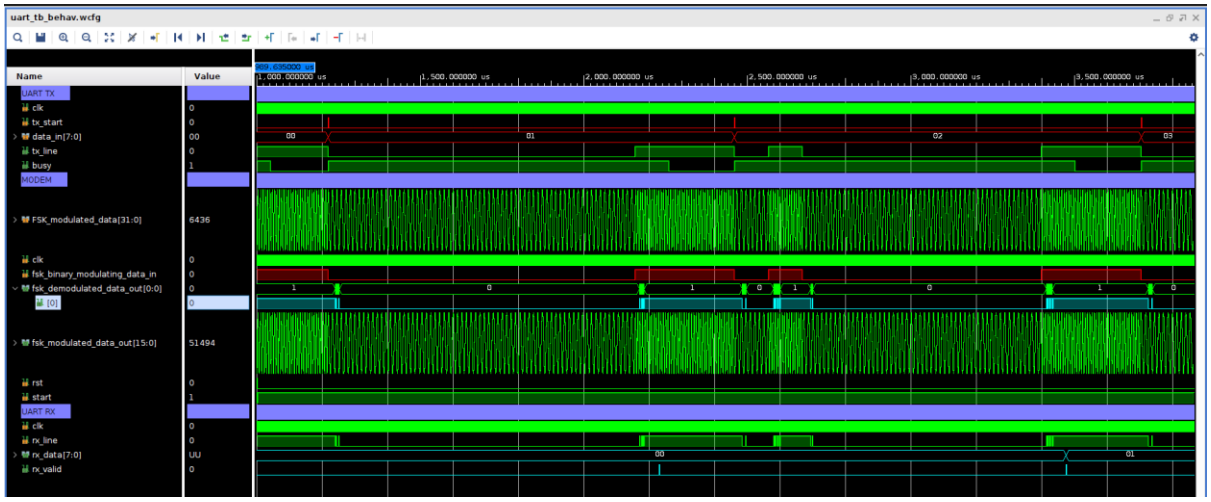
```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity uart_tx is
    Port (
        clk      : in  std_logic;
        tx_start : in  std_logic;
        data_in  : in  std_logic_vector(7 downto 0);
        tx_line  : out std_logic;
        busy     : out std_logic
    );
end uart_tx;

architecture Behavioral of uart_tx is
    constant CLK_FREQ   : integer := 100_000_000; -- Hz
    constant BAUD_RATE  : integer := 9600;
    constant BIT_PERIOD : integer := CLK_FREQ / BAUD_RATE;

    type tx_state_type is (IDLE, START_BIT, DATA_BITS, STOP_BIT);
    signal tx_state   : tx_state_type := IDLE;
    signal tx_counter : integer := 0;
    signal bit_index  : integer range 0 to 7 := 0;
    signal tx_buffer  : std_logic_vector(7 downto 0) := (others => '0');
    signal tx_reg     : std_logic := '1';
begin

    process(clk)
    begin
        if rising_edge(clk) then
            case tx_state is
                when IDLE =>
                    tx_reg <= '1';
                    busy <= '0';
                    if tx_start = '1' then
                        tx_buffer <= data_in;
                        tx_state <= START_BIT;
                        tx_counter <= 0;
                        busy <= '1';
                    end if;

                when START_BIT =>
                    tx_reg <= '0'; -- start bit
                    tx_counter <= tx_counter + 1;
                    if tx_counter = BIT_PERIOD then
                        tx_counter <= 0;
                        tx_state <= DATA_BITS;
                        bit_index <= 0;
                    end if;

                when DATA_BITS =>
                    tx_reg <= tx_buffer(bit_index);
                    tx_counter <= tx_counter + 1;
                    if tx_counter = BIT_PERIOD then
                        tx_counter <= 0;
                        bit_index <= bit_index + 1;
                        if bit_index = 7 then
                            tx_state <= STOP_BIT;
                        end if;
                    end if;

                when STOP_BIT =>
                    tx_reg <= '1'; -- stop bit
                    tx_counter <= tx_counter + 1;
                    if tx_counter = BIT_PERIOD then
                        tx_counter <= 0;
                        tx_state <= IDLE;
                    end if;
            end case;
        end if;
    end process;

    tx_line <= tx_reg;

end Behavioral;
```

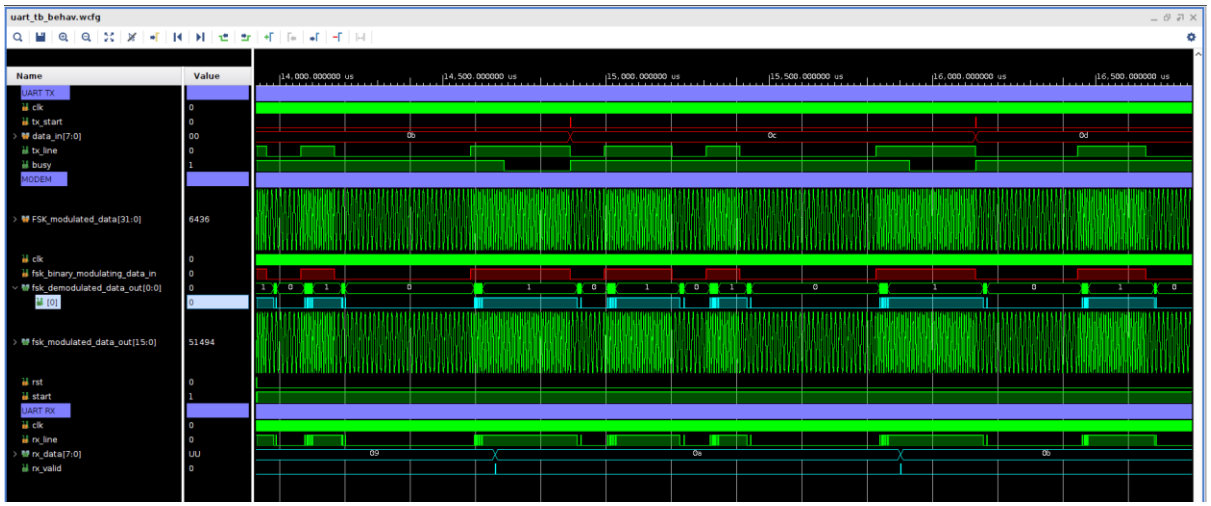
The testbench simulates a full uart transmission and reception path through an fsk modulation-demodulation chain. Here's the breakdown:

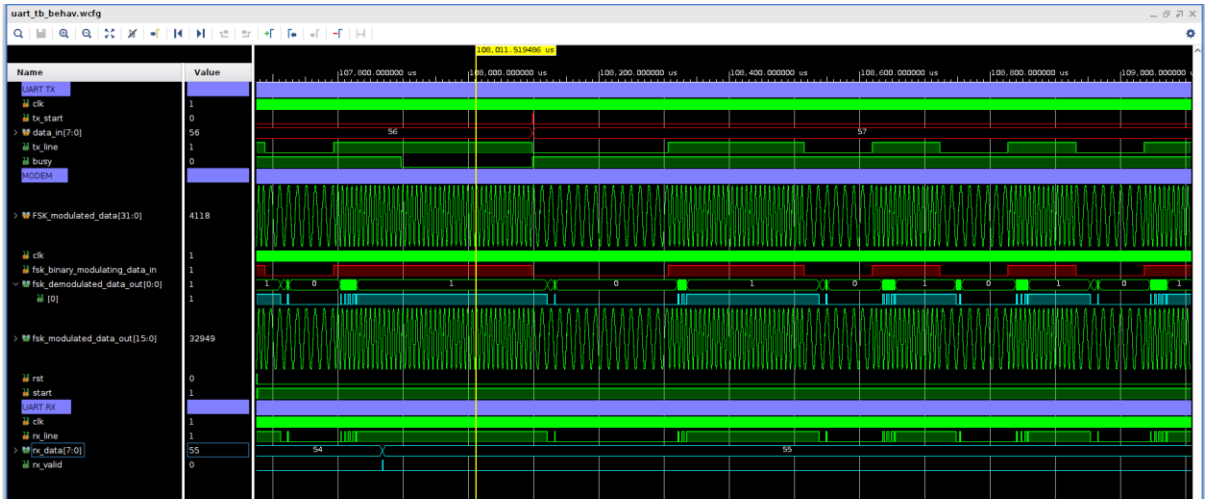
- generates a 100 mhz clock signal,
- instantiates both uart_tx and uart_rx modules,
- feeds uart_tx output into an fsk modem,

- recovers the modulated signal via fsk_demodulated_data_out and feeds it to uart_rx,
- controls the transmit sequence by asserting tx_start and incrementing tx_data,
- resets the system at startup and then enables the fsk modem via start,
- asserts after each reception that tx_data equals rx_data plus 1 modulo 256,
- loops through 1024 transmissions, verifying data integrity,
- ends simulation after all transfers.

By running the testbench, we can see that the assertion does not get triggered, and that the received data is the expected one:







CONCLUSIONS

This tutorial bridges theory and practice in the implementation of an FSK modem using FPGA tools.

By leveraging Simulink for modeling, Vitis HLS for hardware-optimized C/C++ design, and Vivado for FPGA integration, the document walks through a complete development FLOW.

The additional UART integration validates the modem's real-world utility.

With clear modular breakdowns and automation scripts, this project serves as a robust starting point for more complex communication systems on FPGA platforms.