EEC0055 - Digital Systems Design

2018/2019

Laboratory 3

19 November - ~~21 December 2018~~ (new due date: **11 January 2019**)

All-digital FM stereo modulator

Version 0.6 – 27 December 2018

Revision history

date	notes	author
V0.1 Nov 18, 2018	First version	jca@fe.up.pt
V0.2 Nov 23, 2018	Added section 3.2.1 Verification kit for the generic DDS module; Modified section 3.1, changing the specification of the main clock frequency to 147.456000 MHz (integer multiple of 48 kHz);	jca@fe.up.pt
V0.3 Nov 26, 2018	Added section 4 – IP cores, with the description of the two intellectual property blocks provided for the FM modulator implementation: sequential multiplier and 4X interpolator	jca@fe.up.pt
V0.4 Dec 13, 2018	Added section 5 with the description of the verification environment provided (Verilog testbench and Matlab/Octave programs) Added section 6 with updates to the initial specification and additional implementation details The due date has been updated to January 11th, 2019	jca@fe.up.pt
V0.5 Dec 21, 2018	Added section 5.1, 5.2 and 5.3 referring to the updated Verilog and Matlab simulation models and adds two figures 11 and 12, referred in these sections. Section 5.3 includes some additional guidelines for debugging.	jca@fe.up.pt
V0.6 Dec 27, 2018	Added new section 7 with the description of the final implementation and verification project.	jca@fe.up.pt

1 – Introduction

In this project the students will implement a FM stereo modulator and transmitter. The input stereo audio signal is received from an audio source connected to the Line-in or Mic-in inputs of the Atlys board. These analog signals are digitized by an audio codec on the board. Inside the reference project provided for the Atlys board, where your design will be implemented, the digital audio signal is already available as two 18-bit signed words sampled at 48 kHz. The same serial interface implemented in the previous projects will be used in this design to configure some parameters of the FM modulator.

Your design will receive the stereo high-quality digital audio signal, generate the multiplexed FM-stereo signal to modulate in frequency a 5 MHz sinusoidal signal, sampled at 160 MHz. The digital output (8 bits) will be connected to an external digital to analog converter (DAC) that will produce an FM analog signal sampled at 100 Mhz. A FM radio receiver placed nearby the transmitter and tuned to 95 MHz or 105 MHz will be able to receive the FM signal.

Figure 1 shows the overall organization of the system. The FM modulator (block FMSTEREO) is detailed in the next section and a complete Matlab will be provided for generating golden simulation data for each of the blocks.

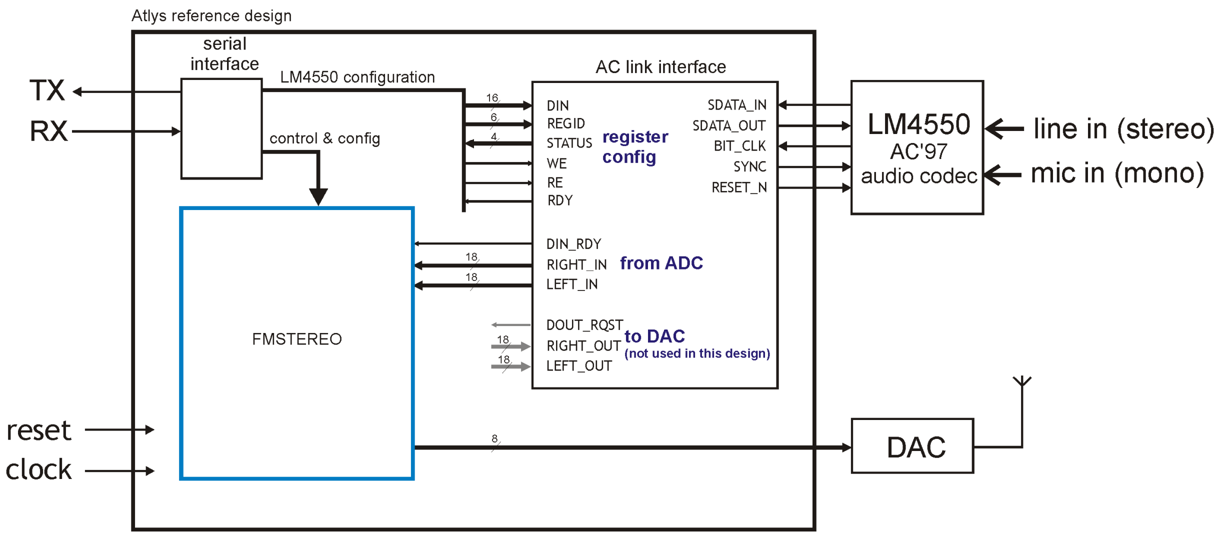

Figure 1 – Simplified block diagram of the reference project. The design to develop is represented by the block FMSTEREO.

2 – FM and FM stereo basics

This section introduces the basics concepts of frequency modulation and the process of encoding the two channels of a stereo signal. If you are familiar with these concepts you can skip this section.

2.1 – Frequency modulation

FM or frequency modulation is a process to encode analog information (a low frequency audio signal (t)) on a high frequency carrier signal (for example the FM broadcast radio signal in the band 88 – 108 MHz). In a FM signal, the instantaneous frequency is a fixed value ω_c (the frequency tuned to listen to a FM radio station) summed to a deviation in frequency that is proportional to the amplitude of the encoded signal $m(t)$. The FM modulation used in the commercial FM broadcast in Europe uses a maximum frequency deviation of ±50 kHz.

The instantaneous angular frequency of a FM signal $\omega_i(t)$ is thus defined as:

$$\omega_i(t) = \omega_c + K_f m(t) \tag{1}$$

where K_f is a frequency deviation constant that depends on the maximum amplitude of the signal $m(t)$.

A sinusoidal signal with constant angular frequency ω_c is represented by:

$$x(t) = A.\cos(\omega_c t) = A.\cos(\theta(t)) \tag{2}$$

where $\omega_c t = \theta(t)$ is the instantaneous angle argument of the cosine function. If the frequency varies with time, as $\omega_i(t)$ in equation 1, the angle argument of the cosine function is the integral with time of that frequency:

$$\theta(t) = \int_{-\infty}^{t} \omega_i(\alpha)\, d\alpha = \int_{-\infty}^{t} (\omega_c + K_f m(\alpha))\, d\alpha \tag{3}$$

The process to generate a FM signal uses directly the relationship in equations 2 and 3: the output signal is the cosine of an angle $\theta(t)$ obtained by accumulating along time a constant ω_c added to the signal to modulate.

2.2 - FM Stereo

A FM stereo broadcast encodes a two channel audio signal by calculating the sum of the two channels $L + R = (left(t) + right(t))$ and the difference between them $L - R = (left(t) - right(t))$. The signal $m(t)$ is obtained by summing $L + R$, $L - R$ multiplied by a 38 kHz sine wave and a 19 kHz pilot sine wave (the constants K_s, K_d and K_p will be defined later).

$$m_{stereo}(t) = K_s(left(t) + right(t)) + K_d(left(t) - right(t)) \times \cos(2\pi \times 38k\, t) + K_p \cos(2\pi \times 19k\, t)$$

Figure 2 shows the spectrum of a FM stereo signal.

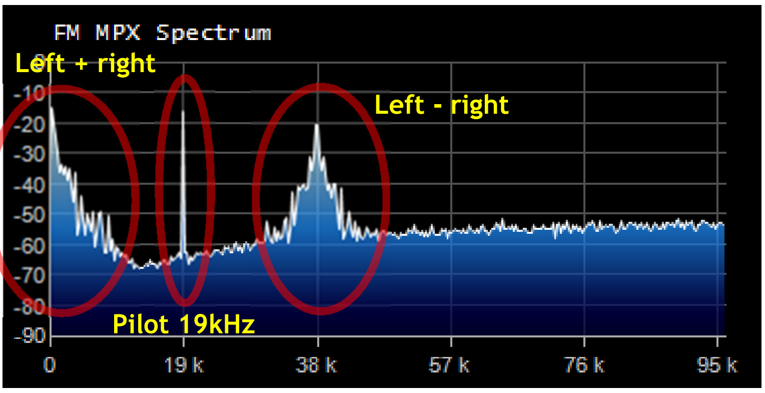

Figure 2 – Spectrum of a FM stereo signal (image created by the software-radio application SDRSharp – free download from airspy.com)

3 - Digital implementation

The construction of the FM stereo modulator requires the implementation of 3 different cosine functions, two with constant frequencies (19 kHz and 38 kHz) and one with the angular argument defined by equation 3. This is thus a fundamental building block for our system and we will start by implementing a parameterizable digital system to generate a sinusoidal wave.

One process to implement a digital calculator of the function cos(x) is called Direct Digital Synthesizer (DDS[1], see figure 3) and is based on a look-up table (a ROM-like memory) holding N equally spaced samples of one period of the function cosine (or any other periodic function). That memory is addressed by an accumulator register whose instantaneous value represents the angle argument of the cosine. The step of increment of the accumulator (the value accumulated at each clock cycle) dictates the frequency of the output cosine function. To generate the FM signal, that step must be proportional to $\omega_c + K_f m(t)$.

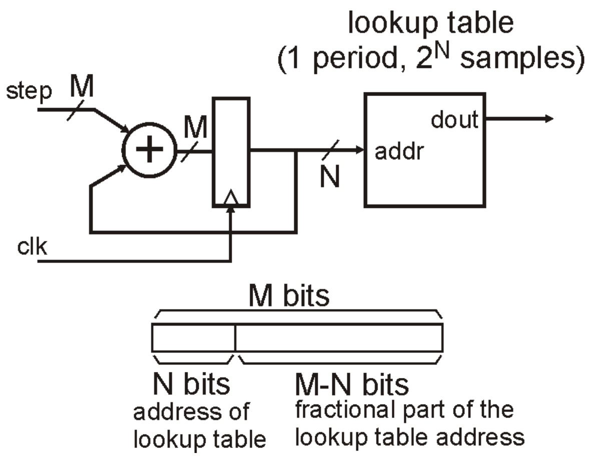

Figure 3 – Direct Digital Synthesizer

The frequency of the output wave is determined by equation 4, where F_{sine} represents the frequency of the output signal and F_{clk} Is the clock frequency.

$$F_{sine} = \frac{F_{clk}}{2^N} \times step \qquad (4)$$

Input **step** is a fixed point value with N integer bits and (M-N) fractional bits, defining the phase increment for each clock cycle.

Figure 4 presents a simplified block diagram of the complete FM stereo modulator. The target clock frequency will be 160 MHz and the whole circuit should be synchronous with the main clock, using a global synchronous reset.

The input signals (the left and right audio channels) are available as 18 bit two's complement words sampled at 48 kHz. Signals **LpR** and **LmR** (addition and subtraction of the two input channels) are multiplied by the constants **Ks** and **Kd** (4 bit unsigned) representing positive numbers between 0 (0.000b) and 1.875 (1.111b). The results of these multiplications are scaled to 18 bits and up-sampled to 4 x 48kHz = 192 kHz using a first-order interpolator (this block will be available as a RTL Verilog module). The **LmR** signal is then multiplied by a 38 kHz cosine function generated by a DDS, using the same 192 kHz sampling frequency. The result of this multiplication is scaled down again to fit into 18 bits. Finally, the **mstereo** signal is constructed by adding the scaled **LpR** signal, the **LmR** multiplied by the 38 kHz cosine and the output of a second DDS generating a 19 kHz cosine function. The output of this addition is scaled down to fit the 20 bit dynamic range.

The FM modulation is performed by a third DDS, where the accumulator increment **Phase** is a linear function of the **mstereo** signal: **Phase = stepWc + mstereo * Kf**. This DDS will have a lookup-table with 32 entries (for the whole cosine period), holding samples with 8 bits in two's complement signed format. With **stepWc**=1 and the clock frequency equal to 160 MHz, the central frequency of the output sine wave will be 5 MHz. The output of this DDS is applied to an external digital to analog converter to generate the output analog signal.

The parameters **Ks**, **Kd**, **Kp**, **Kf** and **stepWc** will be provided by output ports of the serial interface. Additional ports can be used to configure other parameters of your design.

[1] A brief introduction to DDS can be found in ftp://ftp.ni.com/evaluation/pxi/Direct_Digital_Synthesis.pdf

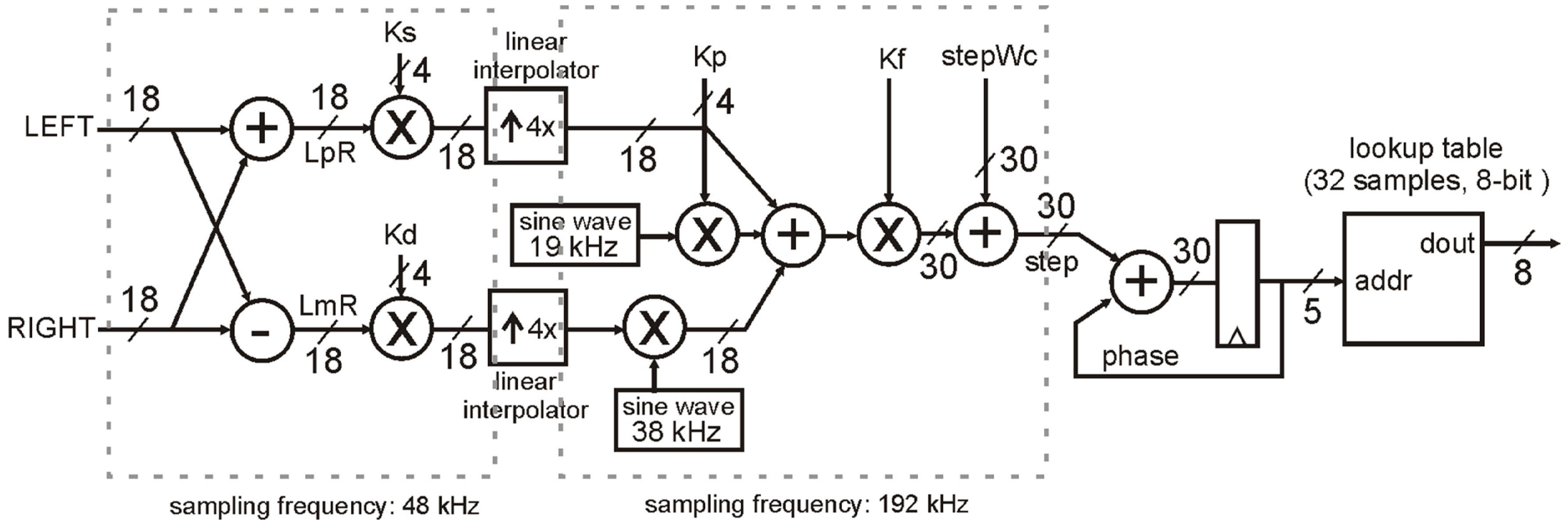

Figure 4 - FM stereo digital modulator

3.1 Design goals and constraints

The global design goal should be to minimize the area, measured as the number of lookup tables and flip-flops. The arithmetic blocks available in the FPGA (DSP48) should not be used in this design. A small sequential multiplier will be available to implement the multiplications shown in the datapath.

The clock frequency of the final DDS should be 147.456000 MHz. The rest of the circuit can use that same clock frequency (the easiest solution) or slower clocks with frequencies adjusted to the requirements of each section. This later option is more challenging as it requires to solve the clock domain crossing (CDC) synchronization issues. Note that a 48 kHz clock would be sufficient for the initial section before the interpolators, but in that case combinational (and large) multipliers need to be used.

3.2 Development stages - generic DDS

The first block to develop is a parameterizable digital sine wave generator, using the DDS architecture referred above. Build a Verilog module implementing the circuit represented in figure 2. The same Verilog module should be flexible enough to implement the 3 DDS blocks needed in the FM modulator. The contents of the lookup table can be loaded at compile time from a text file (either for simulation and synthesis) using the Verilog system task **`$readmemh( )`**. This module must have a clock enable input and a synchronous reset.

3.2.1 Verification kit for the generic DDS module

This verification kit is prepared to perform the logic simulation the Verilog module that implements the DDS sine generator. This is provided as a set of files organized with the same directory structure used for the previous projects. The kit contains:

./matlab/dds.m	Matlab/Octave script to create the simulation data required for the testbench and to analyse the sine wave for different configurations of the DDS parameters;
./simdata	The script dds.m will create in this directory two data files that will be used by the Verilog testbench;
./src/verilog-tb/dds_tb.v	The testbench to perform the functional and the post-synthesis simulation;

To use this kit you must follow exactly the instructions below (this same text is included in the beginning of the testbench dds_tb.v:

1. Edit the script ./matlab/dds.m and adjust the following parameters to match the DDS configuration you want to verify:

```
duration = 0.1;      % Duration of the output test signal (seconds):
Fs = 192000;         % Sampling frequency (Hz):
Fout = 19000;        % required output frequency (Hz):
Nbits_sine_LUT = 9;   % Number of bits per sample in lookup-table
Nsamples_LUT   = 128; % Number of samples in the lookup-table (int power of 2)
Nfrac = 6;           % Number of bits of the fractional phase:
                     %   note that the number of bits in the integer part of the
                     %   phase is given by log2( NsamplesLUT )
```

2. Run the script dds.m in Matlab/Octave, in directory ./matlab. This will generate the data for the DDS lookup table (file ./simdata/DDSLUT.hex) and a vector with the output sine samples generated by the DDS (file ./simdata/DDSout.hex). These files are ASCII with one signed data sample per line in hexadecimal format and the high-order bits to

the left of the Nbits_sine_LUT bit padded with zeros (only the lower Nbits_sine_LUT are meaningful). To use the DDSLUT.hex file to load you lookup-table (register array sineLUT), you can include the following Verilog code into your module:

```
reg [31:0] sineLUT[ 0 : Nsamples_LUT-1 ];
initial
  $readmemh("../simdata/DDSLUT.hex", sineLUT );
```

The lookup-table should be defined as a 32-bit register array, even if the data samples use less bits. The output port of the DDS module should only output the meaningful bits. This script also outputs the required phase increment for the output frequency set.

Register this number as it will be necessary to configure the parameter PHASE_INCREMENT in the testbench

3. Adjust the following simulation parameters in the testbench:

```
parameter FS             = 192000; // Sampling frequency
parameter MAX_SIM_SAMPLES = 19200;  // Maximum simulation time is 0.1 second
parameter N_OUTPUT_BITS   = 9;      // Number of valid bits in the output word
parameter PHASE_INCREMENT = 32'b001100_101010; // 12.6562500 in binary: 001100.101010
                                              // for generate a 19 kHz sine wave
```

4. Setup and run the simulation in QuestaSim:

4.1 Create a QuestaSim project in ./sim

4.2 Import to the project your dds.v and this testbench (./src/verilog-tb/dds_tb.v). You may need to adjust the module and signal names and define the parameters needed to configure your module: number of samples in the DDS LUT, number of bits per sample and number of bits of the fractional part of the phase. Note that the example of instantiation included in this testbench does not define any parameter.

4.3 The testbench compares automatically the results generated by the DDS module with the results generated by the Matlab script. If errors are found and you need to analyse the signals in more detail, the signal 'outsineNbits' contains the output with only the number of bits defined by parameter 'Nbits_sine_LUT'. This signal can be plotted in the waveform window using radix decimal and format analog.

4.4 If no errors are reported for the various configurations needed, congratulations! You have created a fundamental building block of the FM modulator.

5. If the simulation succeed, you can proceed with the RTL synthesis and post-synthesis verification, using this same testbench (refer to the guide of laboratory project 2).

4 - IP blocks

Two IP (*intellectual property*) blocks are made available for free: a signed multiplier and a 4X linear interpolator. These are provided as Verilog RTL synthesizable modules, including very basic testbenches to illustrate the module instantiation and the respective eco systems. These modules are available in directory **./IP-cores/seqmultNM** and **./IP-cores/interpol4x**.

4.1 – Sequential signed multiplier – module seqmultNM (seqmultNM.v)

The sequential signed multiplier implements the shift-add sequential multiplication algorithm in **N+2** clock cycles (**N** is the number of bits of the multiplier). The module is instantiated as:

```
seqmultNM     #(           // definition of parameters
                .N( N ),   // parameter N = number of bits of the multiplier
                .M( M )    // parameter M = numbero of bits of the multiplicand
               )
seqmult_1                  // instance name
          (
           .clock( clock ), // Master clock
           .reset( reset ), // Master reset, synchronous and active high
           .start( start ), // Set to 1 during one clock cycle to start the multiplication
           .ready( ready ), // Set to 1 when the multiplier is ready to accept a new start
           .A( A ),         // Multiplicand, signed  M bits
           .B( B ),         // Multiplier, signed    N bits
           .R( R )          // Result, signed        M+N bits
          );
```

The module receives two parameters to specify the number of bits of the multiplicand (parameter **M**) and the number of bits of the multiplier (parameter **N**). This module requires **N+2** clock cycles to complete one multiplication and the signed result has **N+M** bits. Setting input **start** to 1 during one clock cycle will load the operands **A** and **B** (multiplicand and multiplier, respectively) and start the shift-add iterative process. When the iterative process is running, the output **ready** is set to 0 and returns to 1 when the multiplication ends. The result is ready at output **R** when output **ready** is 1 after an activation of **start.** While the process is running, the output **R** has meaningless data.

Figure 5 shows the simulation waveforms for a multiplication with a 14 bit multiplicand (input **A**) and a 12 bit multiplier (input **B**). The result in output **R** is ready **N+2** clock cycles after **start** is set.

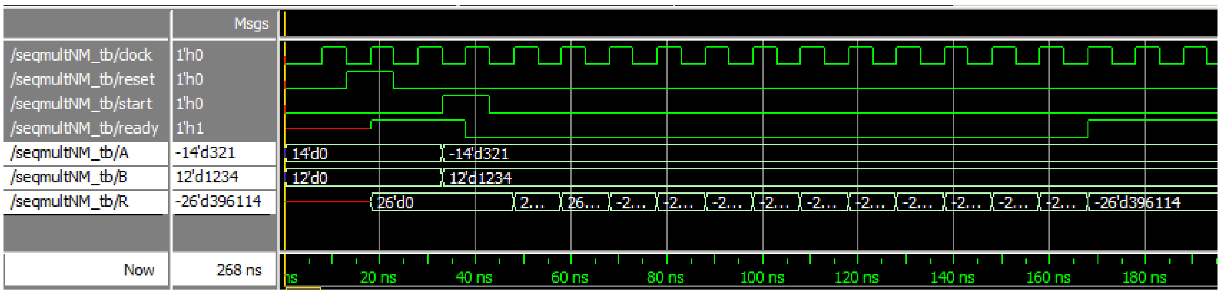

Figure 5 – Example of simulation waveforms for the sequential multiplier.

4.2 - Linear interpolator x4 - module interpol4x (interpol4x.v)

The linear interpolator receives a stream of 18-bit samples (input **xkin**) sampled at a rate defined by the clock enable signal **clkenin** and generates a stream of 18-bit samples (output **ykout**) with a sampling frequency determined by the clock enable signal **clken4x**. The output sampling frequency must be exactly 4 times higher than the input sampling frequency and the additional samples at the output are obtained by the linear interpolation between each two samples of the input stream.

The module can be instantiated as:

```
interpol4x
interpol4x_1(
     .clock( clock ),          // Master clock
     .reset( reset ),          // Master reset, synchronous active high
     .clkenin( en48000Hz ),    // Input clock enable
     .clken4x( en192000Hz ), // Output clock enable (4X the input enable)
     .xkin( xkin ),            // Input signal, 18 bit signed
     .ykout( ykout )           // Output signal, 18 bit signed
     );
```

Signals **clkenin** and **clken4x** are clock enable signals (active only during one clock cycle) whose frequencies dictate the input sampling rate and the output sampling rate. In your design these frequencies will be 48 kHz and 192 kHz.

Figure 6 shows in detail the relationship between the two clock enable signals and figure 7 presents a simulation waveform created with the simple testbench included in the kit. To observe the analog view, configure the signal with radix decimal (signed, two's complement) and format analog (set the maximum and minimum values in the signal and the height of the display window reserved for that signal).

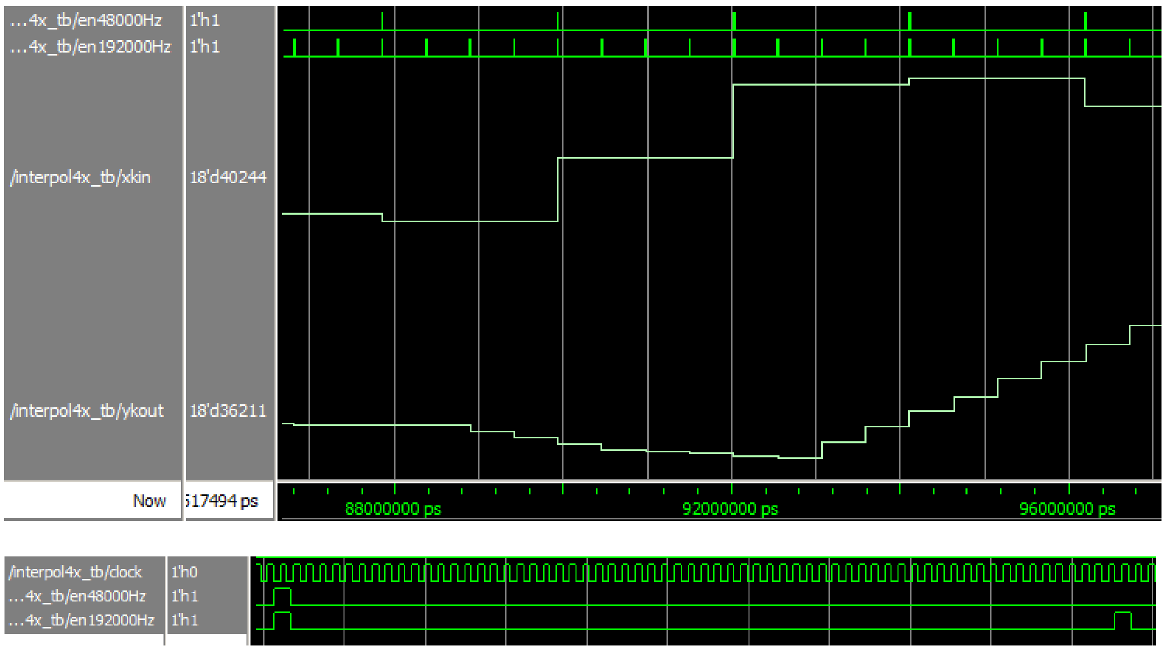

Figure 6 – Example of simulation waveforms for the 4X interpolator – relationship between the two clock enable signals.

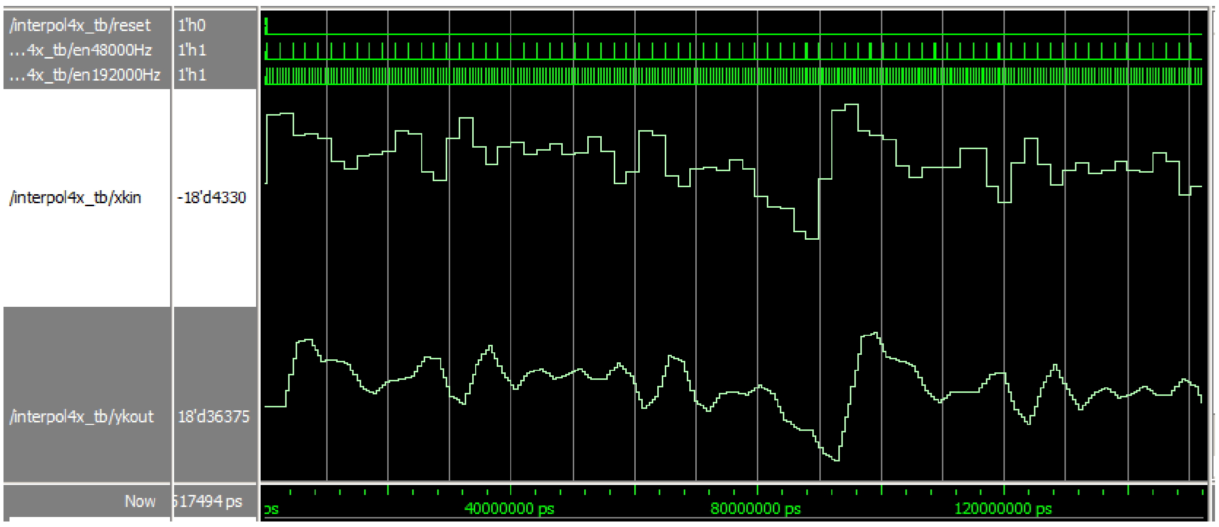

Figure 7 – Example of simulation waveforms for the 4X interpolator – input signal samples at Fs and output signal sampled at 4xFs with linear interpolation.

5 - Verification environment

The verification of the RTL model is done with a self-checking testbench (Verilog) and a set of Matlab/Octave scripts to generate and process various data files required for setting up and running the logic simulation of the RTL model.

This verification environment applies to the signal at the output of the multiplier by the gain Kf (see figure 9 below, signal **FMout**). If you have already implemented the whole system, you should add one additional output to bring out the signal at this point. If you did not yet finish the model, ignore for now the final DDS and the adder with `stepWc`.

The diagram in figure 8 illustrates the verification flow that is further detailed below.

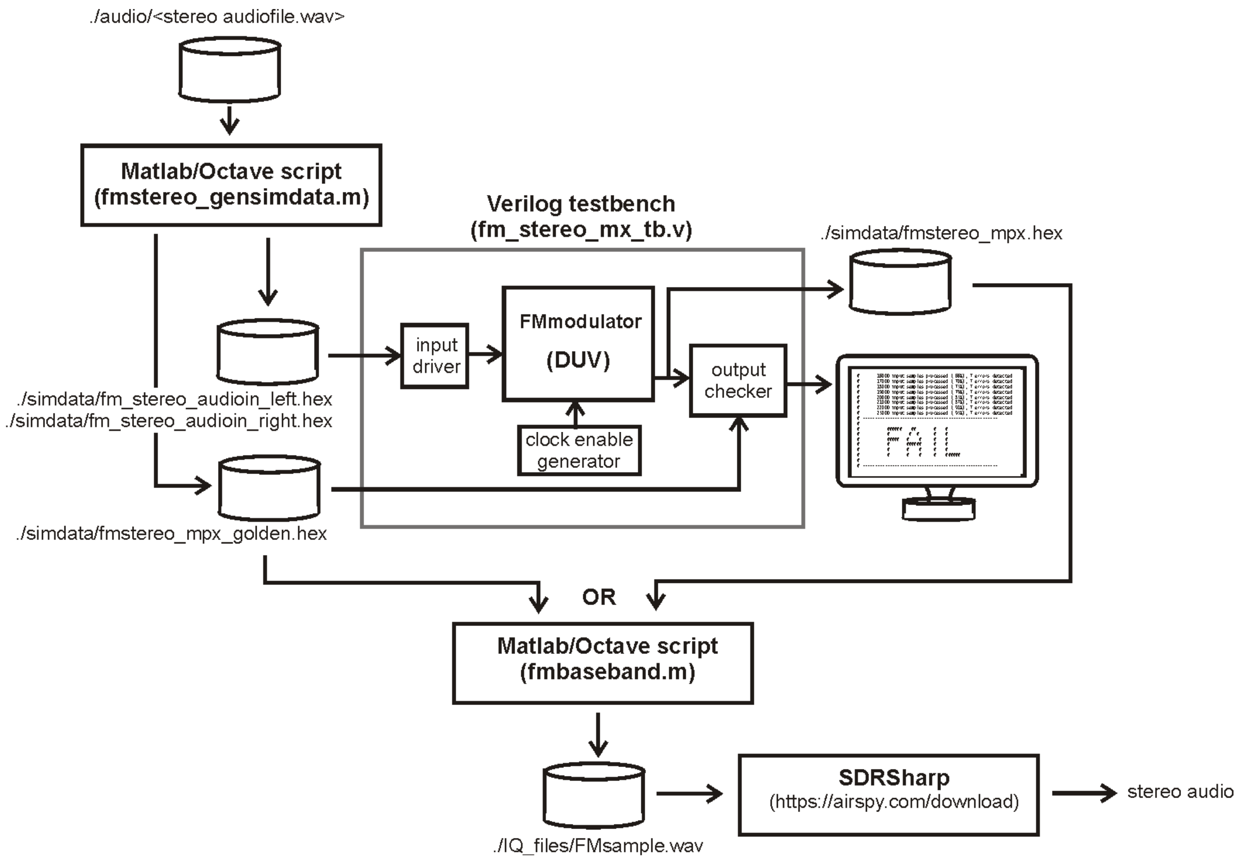
./audio/<stereo audiofile.wav>
Matlab/Octave script (fmstereo_gensimdata.m)
Verilog testbench (fm_stereo_mx_tb.v)
./simdata/fmstereo_mpx.hex
input driver
FMmodulator (DUV)
output checker
clock enable generator
FAIL
./simdata/fm_stereo_audioin_left.hex
./simdata/fm_stereo_audioin_right.hex
./simdata/fmstereo_mpx_golden.hex
OR
Matlab/Octave script (fmbaseband.m)
SDRSharp (https://airspy.com/download)
stereo audio
./IQ_files/FMsample.wav

Figure 8 – Verification flow. The Matlab/Octave script fmbaseband.m implements the final FM modulation performed by the final DDS, not considered for this verification process.

i) Run the Matlab/Octave script `./matlab/fmstereo_gensimdata.m`. To run in Octave you may need to uncomment the statement "`pkg load signal`" (around line 4). Before running the script, you should configure some parameters, between lines 1 and 30, mainly the 4 gains that will be the inputs of your module. Note that these gains must have the bit width referred in the program and must be set accordingly (lines 13 - 16). The data filenames may also be changed (not needed and not recommendable). This program executes the following main tasks:

1. reads a stereo audio file (some audio samples available in `./audio`);
2. resamples the audio data to 48 kHz;
3. generates two text files in hexadecimal format with the audio samples quantized to 18 bits, to be read by the Verilog testbench with the tasks `$readmemh()`. The default filenames for these files are `./simdata/fmstereo_audioin_left.hex` and `./simdata/ fmstereo_audioin_right.hex`;
4. executes a functional model of the FM stereo modulator, using the same parameters specified for the digital implementation: bit widths, truncation of fractional parts, configuration of the DDS modules and interpolator. The final result generated by this program should be equal bit-by-bit to the output of the RTL model;
5. creates a text file with the correct results, in hexadecimal format. The default filename of this file is `./simdata/fmstereo_mpx_golden.hex`.

ii) Edit the toplevel testbench `./src/verilog-tb/fm_stereo_mx_tb.v`, read the comments in the section defining the parameters (lines 1 – 64), configure the simulation parameters and the gains (note the filenames and gains must be the same as in the Matlab/Octave script), and run a simulation with QuestaSim or any other Verilog simulator.

You can run QuestaSim without using the GUI by executing the command "`make`" in directory `./sim` (you need the Cygwin environment that should be already installed in the PCs of the lab). Look at `./sim/Makefile` to understand how to launch a simulation from the command line. File `./sim/vlogfiles` contain the list of Verilog filenames to compile for running the simulation in batch mode. The current file only compiles the source Verilog files provided and uses the pre-compiled modules of the reference FM modulator and the DDS blocks. With these modules you can experiment simulating the testbench and playing with some of the configuration parameters. To simulate with your modules, edit `./sim/vlogfiles` and correct the filenames to match your design hierarchy.

To run QuestaSim with the GUI you can open the simulation project in `./sim/fmmodulator.mpf`. This project only compiles the provided source files and uses the pre-compiled models of the reference FM modulator and DDS blocks.

The testbench reads the two data files generated by the Matlab/Octave program, with the input data samples of the left and right audio signal, applies the samples to the corresponding inputs of the RTL module synchronized with the 48 kHz clock enable signal, captures the output and compares it with the corresponding expected sample read from the golden file created by the Matlab/Octave model.

iii) Run the simulation. The testbench implements a self-checking process and if no error is found nothing more is reported than a final "Pass". If errors are found, the first errors are reported and the simulation is stopped after a certain number of errors is found. Here begins the hard task of debugging your digital design!

5.1 – Probing signals during the Matlab and Verilog simulations

The Matlab/Octave functional model and the Verilog verification kit provides a mechanism to analyze intermediate values along the datapath at each 48 kHz and 192 kHz clock enable cycles. This is useful to facilitate the verification process during the first few input samples and trace the data propagation along the datapath.

To enable this in the Matlab/Octave functional model, set variable **PRINT_DEBUG** to the number of 48 kHz clock cycles to probe (look around line 13 of **fmstereo_gensimdata.m**). Note this number also represents the number of input samples to process, and each input sample will generate 4 set of data in the 192 kHz domain.

To enable this in the Verilog testbench uncomment the definition of symbol **DEBUG_PROBES** and set this symbol to the number of 48 kHz clock cycles to probe (line 19 of **fm_stereo_mx_tb.v**). In the Verilog testbench the signals probed refer to the reference functional simulation model instantiated in the testbench with the instance name DUV_ok (look at the Verilog code between **`ifdef DEBUG_PROBES** and **`endif**, at the bottom of the file). To probe the signals of your model you must adjust the signal names accordingly.

Figures 11 and 12 show the text outputs generated by the Matlab/Octave model and the Verilog testbench for the first 5 input samples (data labels are self-explanatory).

5.2 – Using the Verilog reference model

The updated simulation project for QuestaSim includes a default working library (new library **./sim/work**) with a pre-compiled reference module of a correct FM stereo modulator and the two DDS blocks (modules **dds19k_ok**, **dds38k_ok** and **stereo_fm_mx_ok**). The reference (or *golden*) modules are accurate at the 48 kHz and 192 kHz clock enables, computing the same exact results as the Matlab/Octave functional model (as shown in figures 11 and 12 for the first 5 input samples).

To integrate your design in the testbench it is recommendable to use different module and instance names for the FM modulator and the DDS modules, to preserve the reference models given in the work library. This way, you can compare the outputs of both modules (your and the reference golden modules) at given intermediate points, as referred in section 5.1

5.3 – Suggestions for debugging

To facilitate the verification process you can first verify separately the three datapaths converging to the final adder, by setting only one of the three gains **Ks**, **Kd** and **Kp** to a non-zero value. To obtain the correct final output, the results from these paths must match the expected data at the same 192 kHz clock enable cycles.

If errors are found analyze the beginning of the simulation to look for the cause of the problem (and simulate only for a few set of input samples). To set the first input samples to different values than the created by the Matlab from the audio files you can edit directly the text input files **./simdata/fmstereo_audioin_left.hex** and **./simdata/fmstereo_audioin_right.hex** and set the first few data to values easier to track along the datapath.

Compare the output results of your module with the expected golden data to look for a regular relationship between them that may suggest the origin of the problem. For example, a misalignment of one or two samples along the three datapaths will result in a wrong output but a signal shape (seen as an analog waveform in Questasim) similar to the expected data. Small differences, in the range of few units, may result from incorrect truncation or rounding processes. Create additional variables in the testbench to observe them in the waveform window as an analog wave. This may be useful to identify patterns in the error or correlate the error behavior to other signals.

6 – Updates to the initial specification and additional implementation details

This section updates some specifications and provides additional information about the design implementation. Figure 10 highlights the bus widths of the 192 kHz section.

Clock frequency: the main clock frequency will be equal 48000 x 2048 = 98 304 000 MHz (period 10.173 ns), which is convenient for generating the final FM signal.

DDS modules: the two DDS modules for the 19 kHz and 38 kHz sine waves should be configured with the following parameters (refer to section 3.2.1).

LUT size (one full period): 64 samples
Sample width: 8 bits, signed, two's complement
Fractional part of the phase: 12 bits

Gain Kf: the gain Kf must be 8 bits (unsigned), with 4 bits for the fractional part.

Gain Kp: this gain is 4 bits (the 3 least significant bits represent fractional bits). The 8-bit output of the 19 kHz sine generator must be multiplied by the 4-bit integer Kp and then scaled up to the 18-bit dynamic range by multiplying again by 2^6 (adding 6 zeros at the right).

Multiplication by the 38 kHz sine: the multiplication of the 8-bit output of the 38 kHz sine wave generator by the 18-bit output of the linear interpolator of the left-right path should be scaled down to 18 bits by dividing the result of the multiplication by 2^8.

Final summation: the addition of the three components produces a result that will fit into 20 bits. No scaling is necessary at this point.

Multiplication by gain Kf: the gain Kf is 8 bit wide (unsigned) with 4 bits representing the integer part and 4 bits the fractional part. The result of the multiplication must then discard (truncate) the 4 least significant bits to fit the final result in 24 bits. This is the final result that will be verified by the verification environment described in the section 5.

Clock enable generator: a new module (**`./IP-cores/clockenablegen.v`**) is included that implements a generator of the two clock enables required by the design. This is configurable with two parameters that define the relative timing of the 48 kHz and 192 kHz clock enable pulses (do not modify these parameters unless you know exactly what you are doing and you really need to do that). This module is instantiated in the testbench (between lines 140 and 150) and the Verilog file contains a brief description of the timing between these two signals.

Signed/unsigned multiplications: the sequential multiplier provided implements signed, two's complement multiplications. When multiplying signed data by a word representing unsigned data (as the K gains in this design), this word must be extended by concatenating a zero at the left to represent the same unsigned value but as a positive number. For example, if an unsigned 4-bit word is equal to 1110b its value is 14 decimal but is it is used as a 4-bit operand of a signed multiplier it will be processed as -2 decimal. Adding an extra zero at the left gives 01110b representing +14 decimal as a signed two's complement number.

Truncation of fractional parts: the fractional parts that result from the multiplications by fractional gains (Ks, Kd, Kp and Kf) should always be truncated by discarding the least significant bits. No rounding should be implemented at any point, otherwise the output will not match the result of the Matlab/Octave model and the verification will fail.

Linear interpolator: The module **`./IP-cores/Interpolator-4X/verilog/interpol4x.v`** contains both the previous model and the new implementation (smaller and does not delay the output as with the previous design). The implementation to use by the Verilog compiler is selected by the definition of the symbols **`INTERPOL_MODEL_1`** to select the previous implementation or **`INTERPOL_MODEL_2`** to select the new implementation (this is the default). The current reference model in Matlab only considers the new implementation because the previous design introduced a delay of 6 cycles of the 192 kHz clock enable and this will fail the current verification testbench. If you already used the previous interpolator, it is necessary to compensate this by adding an equal delay at the outputs of the two sine generators. The recommended solution is to use the new model.

Data synchronization and timing along the datapath: The final summation of the three signal components requires that the three operand are correctly synchronized with the sampling clock enable signals, otherwise the verification will fail. To ensure that, the sequence of operations along the datapath should follow the following timing (see figure 9):

i) The input audio samples are applied to the inputs at the rising edge of the main clock when the 48 kHz clock enable is high; These samples must be processed by the L+R and L-R adders and the gain multipliers in less than 2048 clock cycles, as the interpolator will register its inputs at the next 48 kHz clock enable. This way, there is a budget of 2048 clock cycles to perform these operations (using the sequential multiplier) using as little logic as possible.

ii) The output of the interpolator and the outputs of the two sine wave generators will be updated at the rising edge of the clock, at every 192 kHz clock enable pulse. The path from these outputs to the final register shown in figure 9 should take no more than 512 clock cycles (the period of the 192 kHz clock enable signal), as the final register is loaded in a 192 kHz clock enable pulse the result of the outputs produced in the previous clock enable pulse. The clock budget to perform these operations (the datapath section shown in figure 9) is thus 512 clock cycles and you should implement this part also trying to minimize the amount of logic resources.

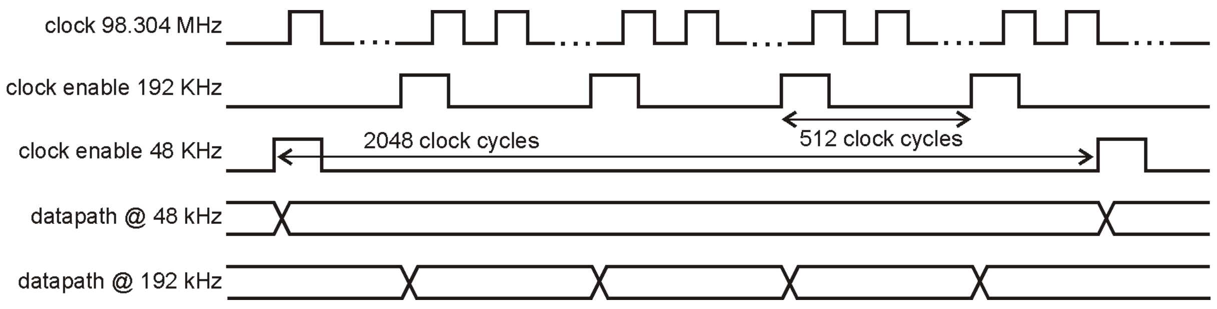

Figure 9 – Timing diagram of the data flow through the datapath.

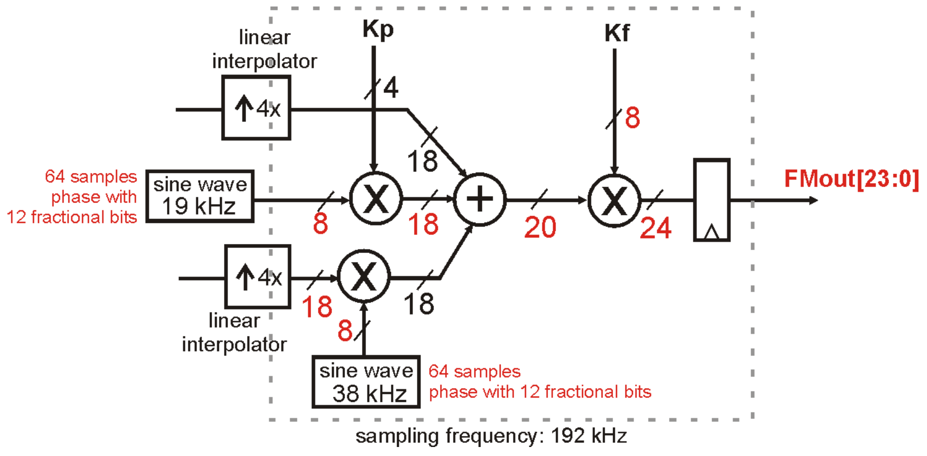
linear interpolator
Kp
Kf
4
8
↑4x
18
64 samples phase with 12 fractional bits
sine wave 19 kHz
8
18
20
24
FMout[23:0]
↑4x
18
8
18
linear interpolator
sine wave 38 kHz
64 samples phase with 12 fractional bits
sampling frequency: 192 kHz

Figure 10 – Updated bus widths of the 192 kHz section. The main output for verification purposes is the signal **`FMout`**. This signal should be the output of a register enabled by the 192 kHz clock enable.

```
48K Cycle   1: L+R=     0  L-R=     0  (L+R)*Ks=     0  (L-R)*Kd=     0
      192K Cycle   1: L+R inter=     0  L-R interp=     0  19k=     0  38k=     0  Pilot=     0  LmR38k=     0  FM=      0
      192K Cycle   2: L+R inter=     0  L-R interp=     0  19k=    71  38k=   117  Pilot= 18176  LmR38k=     0  FM=      0
      192K Cycle   3: L+R inter=     0  L-R interp=     0  19k=   117  38k=    81  Pilot= 29952  LmR38k=     0  FM=  36352
      192K Cycle   4: L+R inter=     0  L-R interp=     0  19k=   125  38k=   -60  Pilot= 32000  LmR38k=     0  FM=  59904
48K Cycle   2: L+R=     0  L-R=     0  (L+R)*Ks=     0  (L-R)*Kd=     0
      192K Cycle   5: L+R inter=     0  L-R interp=     0  19k=    81  38k=  -125  Pilot= 20736  LmR38k=     0  FM=  64000
      192K Cycle   6: L+R inter=     0  L-R interp=     0  19k=    12  38k=   -12  Pilot=  3072  LmR38k=     0  FM=  41472
      192K Cycle   7: L+R inter=     0  L-R interp=     0  19k=   -60  38k=   112  Pilot=-15360  LmR38k=     0  FM=   6144
      192K Cycle   8: L+R inter=     0  L-R interp=     0  19k=  -117  38k=    90  Pilot=-29952  LmR38k=     0  FM= -30720
48K Cycle   3: L+R=    -2  L-R=  -207  (L+R)*Ks=    -1  (L-R)*Kd=  -207
      192K Cycle   9: L+R inter=    -1  L-R interp=   -52  19k=  -125  38k=   -60  Pilot=-32000  LmR38k=    12  FM= -59904
      192K Cycle  10: L+R inter=    -1  L-R interp=  -104  19k=   -90  38k=  -126  Pilot=-23040  LmR38k=    51  FM= -63978
      192K Cycle  11: L+R inter=    -1  L-R interp=  -156  19k=   -12  38k=   -25  Pilot= -3072  LmR38k=    15  FM= -45980
      192K Cycle  12: L+R inter=    -1  L-R interp=  -207  19k=    60  38k=   112  Pilot= 15360  LmR38k=   -91  FM=  -6116
48K Cycle   4: L+R=    -8  L-R=  3559  (L+R)*Ks=    -4  (L-R)*Kd=  3559
      192K Cycle  13: L+R inter=    -2  L-R interp=   734  19k=   112  38k=    98  Pilot= 28672  LmR38k=   280  FM=  30536
      192K Cycle  14: L+R inter=    -3  L-R interp=  1676  19k=   125  38k=   -49  Pilot= 32000  LmR38k=  -321  FM=  57900
      192K Cycle  15: L+R inter=    -4  L-R interp=  2617  19k=    90  38k=  -126  Pilot= 23040  LmR38k= -1289  FM=  63352
      192K Cycle  16: L+R inter=    -4  L-R interp=  3559  19k=    25  38k=   -37  Pilot=  6400  LmR38k=  -515  FM=  43494
48K Cycle   5: L+R=    82  L-R=  4647  (L+R)*Ks=    41  (L-R)*Kd=  4647
      192K Cycle  17: L+R inter=     7  L-R interp=  3831  19k=   -60  38k=   106  Pilot=-15360  LmR38k=  1586  FM=  11762
      192K Cycle  18: L+R inter=    18  L-R interp=  4103  19k=  -112  38k=    98  Pilot=-28672  LmR38k=  1570  FM= -27534
      192K Cycle  19: L+R inter=    29  L-R interp=  4375  19k=  -126  38k=   -37  Pilot=-32256  LmR38k=  -633  FM= -54168
      192K Cycle  20: L+R inter=    41  L-R interp=  4647  19k=   -90  38k=  -127  Pilot=-23040  LmR38k= -2306  FM= -65720
```

Figure 11 - Intermediate signals printed by the Matlab/Octave simulation model (section 5.1)

```
# 48K Cycle   1: L+R=     0  L-R=     0  (L+R)*Ks=     0  (L-R)*Kd=     0
#       192K Cycle   2: L+R->interpol=     0  L-R->interpol=     0  Sin19k=     0  Sin38k=     0  19k x Kp=     0  LmRx38k=     0  FMout=      0
#       192K Cycle   2: L+R->interpol=     0  L-R->interpol=     0  Sin19k=    71  Sin38k=   117  19k x Kp= 18176  LmRx38k=     0  FMout=      0
#       192K Cycle   2: L+R->interpol=     0  L-R->interpol=     0  Sin19k=   117  Sin38k=    81  19k x Kp= 29952  LmRx38k=     0  FMout=  36352
#       192K Cycle   2: L+R->interpol=     0  L-R->interpol=     0  Sin19k=   125  Sin38k=   -60  19k x Kp= 32000  LmRx38k=     0  FMout=  59904
# 48K Cycle   2: L+R=    -2  L-R=  -207  (L+R)*Ks=    -1  (L-R)*Kd=  -207
#       192K Cycle   3: L+R->interpol=     0  L-R->interpol=     0  Sin19k=    81  Sin38k=  -125  19k x Kp= 20736  LmRx38k=     0  FMout=  64000
#       192K Cycle   3: L+R->interpol=     0  L-R->interpol=     0  Sin19k=    12  Sin38k=   -12  19k x Kp=  3072  LmRx38k=     0  FMout=  41472
#       192K Cycle   3: L+R->interpol=     0  L-R->interpol=     0  Sin19k=   -60  Sin38k=   112  19k x Kp=-15360  LmRx38k=     0  FMout=   6144
#       192K Cycle   3: L+R->interpol=     0  L-R->interpol=     0  Sin19k=  -117  Sin38k=    90  19k x Kp=-29952  LmRx38k=     0  FMout= -30720
# 48K Cycle   3: L+R=    -8  L-R=  3559  (L+R)*Ks=    -4  (L-R)*Kd=  3559
#       192K Cycle   4: L+R->interpol=    -1  L-R->interpol=   -52  Sin19k=  -125  Sin38k=   -60  19k x Kp=-32000  LmRx38k=    24  FMout= -59904
#       192K Cycle   4: L+R->interpol=    -1  L-R->interpol=  -104  Sin19k=   -90  Sin38k=  -126  19k x Kp=-23040  LmRx38k=   102  FMout= -63978
#       192K Cycle   4: L+R->interpol=    -1  L-R->interpol=  -156  Sin19k=   -12  Sin38k=   -25  19k x Kp= -3072  LmRx38k=    30  FMout= -45980
#       192K Cycle   4: L+R->interpol=    -1  L-R->interpol=  -207  Sin19k=    60  Sin38k=   112  19k x Kp= 15360  LmRx38k=  -182  FMout=  -6116
# 48K Cycle   4: L+R=    82  L-R=  4647  (L+R)*Ks=    41  (L-R)*Kd=  4647
#       192K Cycle   5: L+R->interpol=    -2  L-R->interpol=   734  Sin19k=   112  Sin38k=    98  19k x Kp= 28672  LmRx38k=   561  FMout=  30536
#       192K Cycle   5: L+R->interpol=    -3  L-R->interpol=  1676  Sin19k=   125  Sin38k=   -49  19k x Kp= 32000  LmRx38k=  -642  FMout=  57900
#       192K Cycle   5: L+R->interpol=    -4  L-R->interpol=  2617  Sin19k=    90  Sin38k=  -126  19k x Kp= 23040  LmRx38k= -2577  FMout=  63352
#       192K Cycle   5: L+R->interpol=    -4  L-R->interpol=  3559  Sin19k=    25  Sin38k=   -37  19k x Kp=  6400  LmRx38k= -1029  FMout=  43494
# 48K Cycle   5: L+R=   830  L-R=  4830  (L+R)*Ks=   415  (L-R)*Kd=  4830
#       192K Cycle   6: L+R->interpol=     7  L-R->interpol=  3831  Sin19k=   -60  Sin38k=   106  19k x Kp=-15360  LmRx38k=  3172  FMout=  11762
#       192K Cycle   6: L+R->interpol=    18  L-R->interpol=  4103  Sin19k=  -112  Sin38k=    98  19k x Kp=-28672  LmRx38k=  3141  FMout= -27534
#       192K Cycle   6: L+R->interpol=    29  L-R->interpol=  4375  Sin19k=  -126  Sin38k=   -37  19k x Kp=-32256  LmRx38k= -1265  FMout= -54168
#       192K Cycle   6: L+R->interpol=    41  L-R->interpol=  4647  Sin19k=   -90  Sin38k=  -127  19k x Kp=-23040  LmRx38k= -4611  FMout= -65720
```

Figure 12 - Intermediate signals printed by the Verilog simulation model (section 5.1)

7 - Design integration and final FPGA implementation

This section describes the complete project where the FM stereo modulator will be integrated and the guidelines for completing the final FPGA implementation. You should proceed to this stage **if and only if** your design has been completed and verified with the functional testbench described in section 5.

This design kit includes a complete project for the Atlys board and a new testbench prepared for performing the 3 simulation tasks (functional, post-synthesis and post-place&route). The self-checking verification included in this testbench only verifies the data at the output of your module, as done by the previous verification environment. The verification of the final output that drives the external DAC is not implemented as it would require a different verification strategy than comparing that to pre-computed golden data (this will be discussed later during the presentation of your projects).

The simulations are now much more complex and slow. In a HP laptop (processor I5 @ 1.6 GHz, 4GB RAM) the average simulation speeds are:

Functional:	1400 input samples/minute (34 minutes to simulate 1 second of audio)
Post-translate:	320 input samples/minute (2.5 hours to simulate 1 second of audio)
Post-P&R:	33 input samples/minute (more than 24h to simulate 1 second of audio)

To conclude the implementation for the FPGA board you must perform the following tasks:

a) Download the project archive for the XILINX ISE tool and expand it in a different folder than your current project directory (the directory tree keeps the same structure as before).

b) Copy your Verilog source files that form your project to the directory **./src/verilog-rtl/Alunos/name1-name2** ("name1-name2" should be your short names, as "joao-joana"). Do not copy the sources of the interpolator, multiplier and clock enable generator as these are already integrated in the project and referred to their location in the **./IP-cores** folder.

c) Create a new Verilog file in that directory, named **instance.v**, with the instantiation of your top module. The inputs and outputs must be connected to the signals shown in the example below (in blue). Name the instance "DUV"

```
my_fm_stereo  DUV
(
      //-----------------------------------------------
      // Global signals
      .clock( clock98MHz ),   // master clock, active in posedge
      .reset( reset ),        // master reset, synchronous, active high

      //-----------------------------------------------
      // Gains:
      .Ks( Ks ), // 4 bits unsigned
      .Kd( Kd ), // 4 bits unsigned
      .Kp( Kp ), // 4 bits unsigned
      .Kf( Kf ), // 8 bits unsigned

      //-----------------------------------------------
      // Audio data in:
      .LEFTin( LEFT_inf ),          // data in, left channel, 18 bits signed
      .RIGHTin( RIGHT_inf ),        // data in, right channel, 18 bits signed

      .clken48kHz( clken48kHz ),    // Clock enable for input sampling rate:
      .clken192kHz( clken192kHz ),  // Clock enable for 4X sampling rate:

      //-----------------------------------------------
      // FM Stereo dataout:
      .FMout( FMout )               // data out, FM stereo signal, 24 bits signed

);
```

d) Edit the toplevel design **./src/verilog-rtl/s6base.v** and around lines 18-20 comment the definition of symbol **REFERENCE_MODEL** and set the symbol **MY_DESIGN** to the path of your **instance.v** file:

```
`define  MY_DESIGN    "../src/verilog-rtl/alunos/name1-name2/instance.v"
```

Your design is instantiated near line 477, with an **`include** directive using this symbol to define the pathname to your file. The project includes a synthesized model of a reference design (file **./src/verilog-netlist/stereo_fm_mx_golden_synthesis.v**), that is instantiated if the symbol **REFERENCE_MODEL** is defined.

e) Run the ISE Project Navigator and open the project file **./impl/lab3.xise**. This will open the XILINX design environment with the complete design for the Atlys FPGA board (refer to the guide of the laboratory project 2 to remember the instructions for running the implementation)

f) Add to the ISE project your Verilog source files in **./src/verilog-rtl/Alunos/name1-name2** and expand all hierarchy levels to check the structure of your design.

g) Run a functional simulation, launching QuestaSim from the ISE environment. First you should run the Matlab script **./matlab/fmstereo_gensimdata.m** to generate the golden simulation results and the input files (as in the previous testbench). Then, edit the new testbench **./src/verilog-tb/s6base_tb.v** and around line 80 set the K gains to the same values used in the Matlab script to generate the simulation data. The value of the parameter **FREQ_STEPWC** is not relevant here and should be kept as is. The text interface is similar and the simulation now opens the waveform window to show some of the relevant signals. This will only be useful if the simulation fails (but it is not supposed to fail!), to analyze the logic signals in detail.

h) In ISE select the toplevel design **s6base** and execute "Synthesize – XST". Look at the synthesis report and correct any error that may appear (look for the latch inference warnings - code 747).

i) Run the Post-translate simulation (post-synthesis) and analyze the results.

j) In ISE select again the implementation flow and run "Implement design". Check the static timing report to verify if the timing constraints were met. Open the timing analysis report: "**Implement design -> Place & route -> Generate … static timing -> Analyze … static timing**". Near the bottom of the text report look for a table with the timing constraints. If you find below that table the text "**All constraints were met.**", your implementation meets the clock frequency requirements. If not, you should analyze further the timing report to identify where the critical paths are, and correct them.

k) Run the post-route simulation. This is the final verification stage and if it succeeds that the design is ready to be implemented and tested in real silicon (in this case, the FPGA board). As there are not enough kits for everybody, the final implementation and experimentation will be done individually, with the assistance of the professors.

7.1 – The top-level design

Figure 13 presents a simplified block diagram of the toplevel design. The gray blocks are implemented as functional processes declared the top level and represent simple "glue logic". The top-level design implements the same serial interface used in the laboratory project 2. This is the primary user interface to program several registers of the audio CODEC, modify the K gains and the output frequency in real time, and control any other design parameter that may be needed to your project.

The digital audio data is provided to your block by a controller that implements a serial to parallel interface with the external audio CODEC (LM4550, datasheet in ./doc/LM4550.pdf). The audio data goes through a set of multiplexers controlled by some of the slide switches to swap the L and R channels or mute them. An additional switch (sw6) selects the signal to drive the final sine synthesizer as the output of the FM modulator (signal **FMout**) or a 440 Hz sinusoidal test signal. The output of the FM modulator is also routed to primary FPGA outputs in the VHDC connector, as well as the two clock enable signals. This is needed only for verification purposes as nothing will be physically connected to these pins. The 7 bit output of the final sine synthesizer in then routed to the PMOD connector (black connector at the bottom left of the board), where the external DAC will be attached.

The testbench uses a functional model of the audio CODEC that implements the parallel to serial and serial to parallel interface of the audio CODEC (**LM4550_digital_sim.v**). The testbench applies the input data samples to this model (simulating the analog inputs), and this module sends them serially to the audio CODEC interface implemented in the FPGA (**LM4550_controler.v**). This controller converts the serial stream to parallel words with the Left and Right audio samples.

Open the top level model **s6base.v** and the testbench **s6base_tb.v** and identify the main blocks and signals shown in figure 13 as well as the different processes and tasks implemented in the testbench.

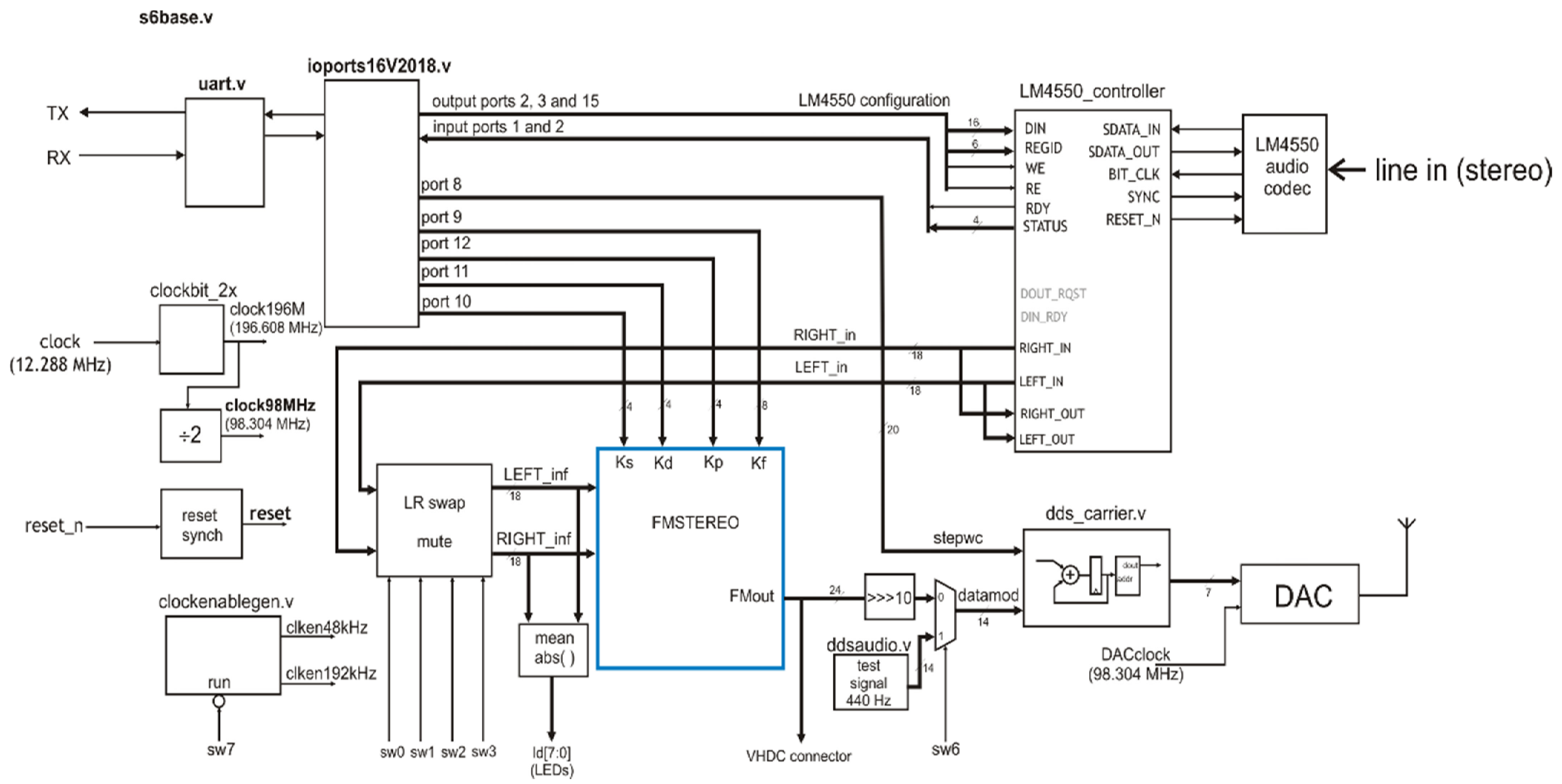

Figure 13 - Simplified block diagram of the top level design (section 7)