VLSI Lab 4

Nicky Advokaat - 0740567 - n.advokaat@student.tue.nl
Marcel Moreaux - 0499480 - m.l.moreaux@student.tue.nl

4$^{\text{rd}}$ quartile, 2014

Abstract

This report contains solutions for the problems described in Assignment L4 for the course VLSI Programming. We will create an upscaler filter.

Contents

1 Problem Specification and Requirements

The goal of this assignment is to construct an upscaler that increases the sample rate of an input signal from 44.1 KHZ to 48 KHZ, which gives the upscaling ratio $\frac{160}{147}$. This is done by first upscaling by L=160, then applying a filter, and the downscaling by M=147. The coefficients for the filter are constructed from the `lanczos` function, which is a finite window version of the `sinc` function. We will have to generate these coefficients ourselves, and store them in ROM memory. There several alternatives to construct the upscaler, these are discussed in section 4.3. The upscaler must have a clock frequency of at least 100 MHz. We have a choice whether to optimize the upscaler for minimum resource utilization, or for maximum throughput. After we have designed and implemented the upscaler we will test it by analyzing the input and output signals.

2 Solution

2.1 Idea

In this section we describe the key ideas behind our design, and the decisions we made during the design process.

First we had to decide which design option of the 4 given options we were going to use. As seen in our answers to question 3 in section 4.3 a naive scaler is very inefficient, it is actually not possible to implement such a scaler on the Xilinx boards we use in the lab. So we will implement the scaler using the direct equation. Now we have to decide between doing the scaling directly, or composing it of multiple scalers. The trade-of here is that a direct scaler has to store more coefficients on the board, whereas a composed scaler uses more multiplications per output. Also, for each scaler we introduce in a composed scaler the delay of the system is increased. We decided to use one direct scaler from 44.1 KHz to 48 KHz.

The next choice we had to make was whether to optimize the scaler for resource usage or for throughput. We decided to optimize for throughput. The most important implication this choice has is that we will use 4 parallel multipliers for the FIR, instead of doing 4 sequential multiplications on one multiplier. We can now compose a diagram of the architecture of our system, figure 1. The coefficients are stored in read only memory (ROM), not visible in the diagram. Some additional hardware is required to get the right coefficient `h[i]` our of the ROM.

To generate the actual coefficients we wrote a small Java program that given some `L` spits out all coefficients in 16 bit signed hex notation. We scaled the coefficients such that they sum up to $(2^{15} - 1) \cdot 4 = 131068$, this way the output signal will have the same intensity as the input signal.

Another observation we made is that since the `lanczos` function is symmetric, the coefficients are symmetric as well. Therefore we could optimize our program by only storing half the coefficients, more precisely we only store $2L + 1$ coefficients instead of $4L$. After implementing this we noticed that as well as using less registers, the post PAR static timing report showed an increase in clock frequency. This can be explained by a more efficient routing because we use less wires.

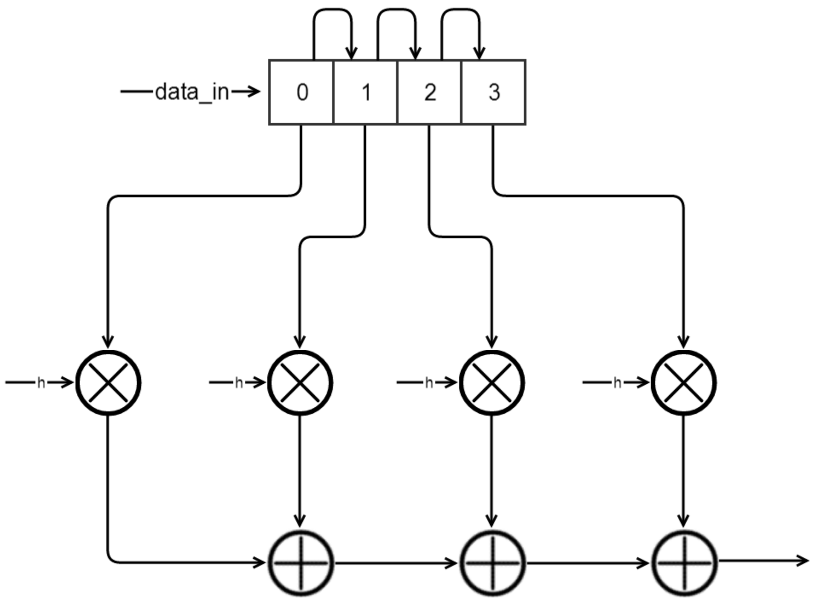

Figure 1: Architecture diagram of the scaler.

2.2 Implementation

In this section we will explain functional correctness of our code. The Verilog source code of our filter can be found in section 5.

reg [0:L_LOG-1] l;

We use the direct equation, this register store the value of $nM \bmod L$, so we can calculate it using only conditionals and adders.

reg signed [0:DWIDTH-1] in [0:3];

For the direct equation we need to store the 4 most recent input values.

reg signed [0:DWIDTH-1] h [0:2*L];

We need to store our coefficients. As explained in the previous section, the coefficients are symmetric. So we can do an optimization storing only $2L + 1$ coefficients.

reg signed [0:DDWIDTH-1] partial1;
reg signed [0:DDWIDTH-1] partial2;
reg signed [0:DDWIDTH-1] partial3;
reg signed [0:DDWIDTH-1] partial4;

We pipelined the system by adding a delay between each of the 4 multiplier and the accumulator of the FIR, this increases throughput.

```
$readmemh(”coefficients.txt”, h);
```

Our coefficients are read from a file. These values are precalculated by a small Java program.

```
in[0] <= data_in;
in[1] <= in[0];
in[2] <= in[1];
in[3] <= in[2];
req_in_buf <= 0;
```

As seen in figure 1, all values in the input buffer are shifted, the new input is loaded into the first.

```
if( l < L - M )
begin
    l <= l + M;
end
else
begin
    l <= l - (L - M);
    req_in_buf <= 1;
end
```

This is used to efficiently calculate the value of $(l + M) \bmod L$. Instead of applying the mod operation each time, we store the value, and increment in by $l+M$. When this value becomes larger or equal to $L - M$ we have to subtract $L - M$.

```
partial1 <= in[0] * h[l]; // h[l+L*0]
partial2 <= in[1] * h[l+L]; // h[l+L*1]
partial3 <= in[2] * h[L*2-l]; // h[l+L*2], mirrored
partial4 <= in[3] * h[L-l]; // h[l+L*3], mirrored
sum <= partial1 + partial2 + partial3 + partial4;
```

This is the filtering part, the multiplications and additions are done in separately because we pipelined the system by buffering between them.

3 Results

3.1 Resource Usage

This section describes some of the resources that our design uses according to the synthesis report.

`321x16-bit dual-port Read Only RAM`

The ROM is used to store our coefficients, we have $2 \cdot L+1 = 321$ coefficients, each 16 bit long.

`16x16 bit multiplier:  4`
There are four multipliers in the filter.

`32 bit adder:  3`

There are three adders to add the 4 outputs of the multipliers.

`8 bit addsub:  1`

For the expressions $l <= l + M$ and $l <= l - (L - M)$, which can not happen during the same clock cycle.

`9 bit adder:  1 9 bit substractor:  2`

3.2 Properties

The sample frequency of the filter is as follows:

- Synthesis report
  - Minimum period: 8.653ns
  - Maximum Frequency: 115.571MHz
- Post-PAR static timing report
  - Minimum period: 8.711nsns
  - Maximum frequency: 114.797MHz

The sample frequency is lower in the static timing report, because it takes into account how the circuit will actually be laid out on the FPGA. The important thing here is that the sample frequency meets the 100 MHz requirement.

3.3 Analysis of Filter Output

This section describes the actual effect of the filter on the input audio file. The filter is an upscaler, so besides increasing the sample frequency it should have as little effect as possible on the signal.

Figure 2 shows the audio signal before and after filtering. The filtered signal has a small delay, but its shape is near identical to the original signal. The plot also shows the startup noise of the filter; the first part of the filtered signal is incorrect. The strength of both signals is about equal, indicating that our filtering coefficients indeed sum up to 1.

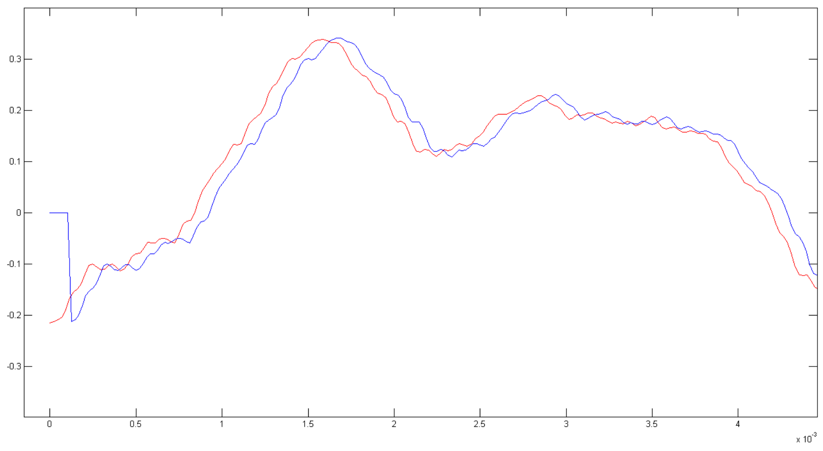

Figure 2: Plot of the first part of the original signal (red) and the signal after filtering (blue).

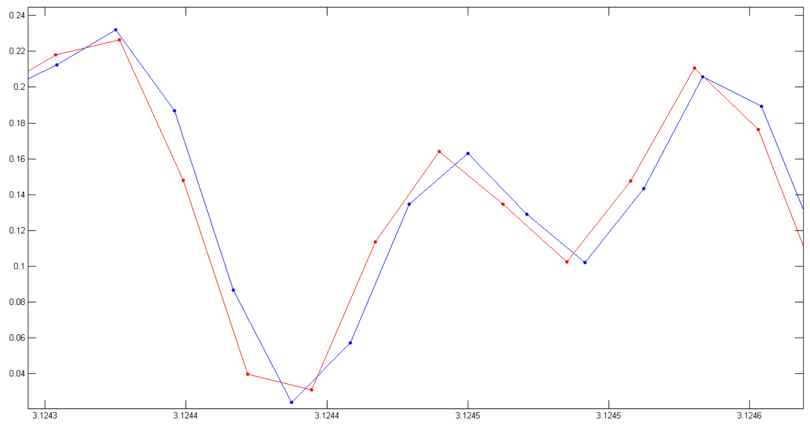

Figure 3: Plot of part of the original signal (red) and the signal after filtering (blue) with dots indicating the samples.

In figure 3 we have zoomed in slightly more, shifted the output signal by 4 samples, and drawn dots at the samples. It shows that the sample frequency of the filtered signal is slightly higher than that of the original signal, while their signal values at each time are roughly the same.

Figure 4 shows the input and output signal in the frequency domain. There is barely any difference between the two spectra, except for some high frequency harmonics in the output signal. These are explained by the fact that the filtered signal has a higher sample frequency, and therefore the maximum frequency it can represent is higher. But this difference is so small that it does not have any audible effect on the filtered audio.

4 Appendix A: Answers to inline questions

This sections contains solutions to the inline questions stated in the assignment description.

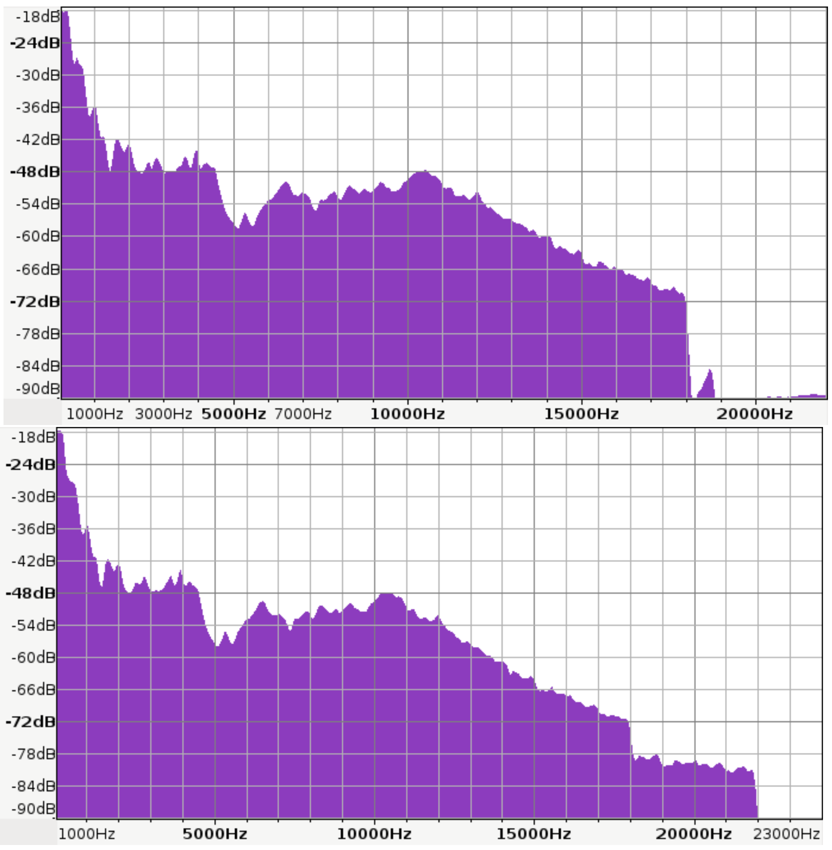

Figure 4: Plot in the frequency domain of the original signal (top) and the signal after filtering (bottom).

4.1 Question 1

$$y[n] = z[Mn]$$

$$z[n] = \sum_{0\le j<4L} h[j]\cdot q[n-j]$$

$$q[n] = \begin{cases} x[n \text{ div } L] & \text{if } n \text{ mod } L = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$y[n] = \sum_{0\le j<4L} h[j]\cdot q[nM-j]$$

$$y[n] = \begin{cases} \sum_{0\le j<4L} h[j]\cdot q[(nM-j) \text{ div } L] & \text{if } (nM-j) \text{ mod } L = 0 \\ \sum_{0\le j<4L} h[j]\cdot 0 & \text{otherwise} \end{cases}$$

The *otherwise* case is always 0 so doesn't contribute to the sum. We can continue with just:

$$y[n] = \sum_{0\le j<4L} h[j]\cdot q[(nM-j) \text{ div } L]) \qquad \text{if } (nM-j) \text{ mod } L = 0$$

$(nM-j)$ mod $L = 0$ happens at:

- $j = nM \text{ mod } L$
- $j = nM \text{ mod } L + L$
- $j = nM \text{ mod } L + 2L$
- $j = nM \text{ mod } L + 3L$

So four times in every summation. If we let j run from 0 to 4 (excl), we get $h[nM \text{ mod } L + jL]$.

$$y[n] = \sum_{0\le j<4} h[nM \text{ mod } L + jL]\cdot q[(nM - j\cdot L) \text{ div } L]$$

$$y[n] = \sum_{0\le j<4} h[nM \text{ mod } L + jL]\cdot q[(nM) \text{ div } L - j]$$

Which is the equation from the assignment.

4.2 Question 2

The period is L.

4.3 Question 3

We assume all values for $h[]$ are precomputed.

- Naive scaler directly
  - Multiplications. The number of coefficients is $4L$. For each sample we need the multiply for each coefficient. The frequency of the FIR is 147 times the output sample frequency. So the number of multiplications per output is $147\cdot 640 = 94080$.

  - Coefficients. $4L = 4 \cdot 64 = 640$.
  - Highest sample rate. $94080 \cdot 48$ kHz ≈ 4.516 GHz assuming sequential FIR implementation.

- Naive scaler composed
  - Multiplications. We have $\frac{M_2}{L_2} \cdot M_1 \cdot (4 \cdot L_1) + M_2 \cdot (4 \cdot L_2) = \frac{14}{15} \cdot 63 \cdot (4 \cdot 64) + 14 \cdot (4 \cdot 15) =$ 15892.8 multiplications per output.
  - Coefficients. $4 \cdot L_1 + 4 \cdot L_2 = 4 \cdot 64 + 4 \cdot 15 = 316$.
  - Highest sample rate. The second sub filter has the highest sample rate at $(4 \cdot 64) \cdot 63 \cdot \frac{14}{15} \cdot 48$ kHz ≈ 722 MHz , using sequential FIR.

- Equation scaler directly
  - Multiplications. In the direct formula the term jL can be precomputed since $0 \leq j < 4$. The term nM does not need a multiplier because n is incremented by one each time, so we can just add M to a stored value. For the same reason the *mod* and *div* do not need multipliers. In the end this only leave the multiplication for $h[] \cdot x[]$. This is done for each j over the summation, so we need 4 multiplications.
  - Coefficients. We need $4L = 640$ coefficients.
  - Highest sample rate. The filter runs at 4 times the output sample rate, $4 \cdot 48$ kHZ $= 192$ kHz.

- Equation scaler composed
  - Multiplications. Using the same arguments as with the direct equation scaler we get $4 + 4 \cdot 14/15 \approx 7.7$ multiplications per output.
  - Coefficients. $4 \cdot 64 + 4 \cdot 15 = 316$.
  - Highest sample rate. The second FIR runs at 4 times the output sample rate, $4 \cdot 48$ kHZ $= 192$ kHz.

5 Appendix B: Verilog source code

This appendix includes Verilog source code for the `filter.v` file in the ISE project.

```
‘timescale 1ns / 1ps

module filter
    #(parameter DWIDTH = 16,
        parameter DDWIDTH = 2*DWIDTH,
        parameter L = 160,
        parameter L_LOG = 8,
        parameter M = 147,
        parameter M_LOG = 8,
        parameter CWIDTH = 4*L)
    (input clk,
        input rst,
        output req_in,
        input ack_in,
        input [0:DWIDTH-1] data_in,
        output req_out,
        input ack_out,
        output [0:DWIDTH-1] data_out);

    // Output request register
    reg req_out_buf;
    assign req_out = req_out_buf;

    // Input request register
    reg req_in_buf;
    assign req_in = req_in_buf;

    // state counter. l = nM mod L (calculated efficiently using
    // conditionals and addition/subtraction)
    reg [0:L_LOG-1] l;
    // The last 4 input samples used in the FIR
    reg signed [0:DWIDTH-1] in [0:3];

    // The FIR coefficients (lots of them)
    // Naively, we'd need [0:4L-1], but since the coefficients are
    // symmetric around h[2L], we can just reuse h[2L-1:1] for // h[2L+1:4L-1]
    reg signed [0:DWIDTH-1] h [0:2*L];

    // The 4 products in the sum are buffered as partial results for pipelining
    reg signed [0:DDWIDTH-1] partial1;
    reg signed [0:DDWIDTH-1] partial2;
    reg signed [0:DDWIDTH-1] partial3;
    reg signed [0:DDWIDTH-1] partial4;
    // Accumulator (lower bits assigned to output port directly)
    reg signed [0:DDWIDTH-1] sum;
    assign data_out = sum >> 15;

    initial
    begin
        // Initialize the ROM with coefficients from file
        $readmemh("coefficients.txt", h);
    end

    always @(posedge clk)
    begin
        // Reset => initialize
        if (rst)
        begin
            req_in_buf <= 0;
            req_out_buf <= 0;
            sum <= 0;
            l <= 0;
        end
        // !Reset => run
        else
        begin
            // Input
            if (req_in && ack_in)
            begin
                // Read new input sample, and shift older samples
                in[0] <= data_in;
                in[1] <= in[0];
                in[2] <= in[1];
                in[3] <= in[2];
                // Input is done
                req_in_buf <= 0;
            end

            // Output
            if (req_out && ack_out)
            begin
                // Output is done
                req_out_buf <= 0;
            end

            // No I/O, computation cycle
            if (!req_in && !req_out && !ack_in && !ack_out)
            begin
                // l <= (l + M) mod L, implemented with conditionals
                // No-overflow case
                if( l < L - M )
                begin
                    l <= l + M;
                end
                // Overflow case. Overflow also means we need a new input!
                else
                begin
                    l <= l - (L - M);
                    req_in_buf <= 1;
                end

                // We output after every computation cycle
                req_out_buf <= 1;

                // pipelined: sum <= (in[0] * h[l] + in[1] * h[l+L] +
                //                    in[2] * h[l+L*2] + in[3] * h[l+L*3]);
                partial1 <= in[0] * h[l];      // h[l+L*0]
                partial2 <= in[1] * h[l+L];    // h[l+L*1]
                partial3 <= in[2] * h[L*2-l];  // h[l+L*2], mirrored
                partial4 <= in[3] * h[L-l];    // h[l+L*3], mirrored
                sum <= partial1 + partial2 + partial3 + partial4;
            end
        end
    end

endmodule
```