VLSI Lab 5

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

4rd quartile, 2014

Abstract

This report contains solutions for the problems described in Assignment L5 for the course VLSI Programming.

Contents

1 Problem Specification and Requirements

We need to implement an upscaler that can process n streams at once, where all streams have the same sample rate and the same upscaling factor. This means we can reuse most of our code from L4 in which we created a single stream upscaler. The calculations performed for each stream are the same as for a single stream, so we just need to store more inputs and interleave the filtering. The coefficients and the way they are stored are equal can be copied from assignment L4 as well. The upscaler has the following requirements:

- The system must run at 100 MHz.
- The system can handle at least 128 streams, but preferably more. We will incrementally test the filter on the number of input streams.
- All streams are correctly upscaled from 44.1 kHZ to 48 kHZ and outputted in the correct order.

2 Solution

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

2.1 Idea

Figure 1 shows an architecture diagram of our design.

To store the last 4 input values for each of the streams, we allocate 4 arrays of size n in BRAM. When input is ready we move the contents of the i^{th} array to the $(i+1)^{th}$ for $0 \leq i < 4$, and store the input value corresponding to jth stream to the j^{th} block of array 0. The filtering happens as follows. For $0 \leq i < n$ and $0 \leq j < 4$ we load the i^{th} value from the j^{th} array from the input buffer arrays. These 4 values are used as input values for the FIR, together with the corresponding coefficients according to the direct equation for the filter. The values from the array are requested one clock cycle before they are needed in the computation, since reads in the BRAM are performed synchronously. As seen in the diagram, we use 4 multipliers in parallel. Alternatively we could have used one multiplier doing 4 sequential operations for each sample, but we decided to optimize for throughput and not for hardware usage.

The coefficients $h[]$ are the same as in lab L4 and because they are symmetric we store only $2L + 1$ of them.

The input storing and FIR processing for each stream happens in the same clock cycle, so that we could theoretically produce one output per clock cycle.

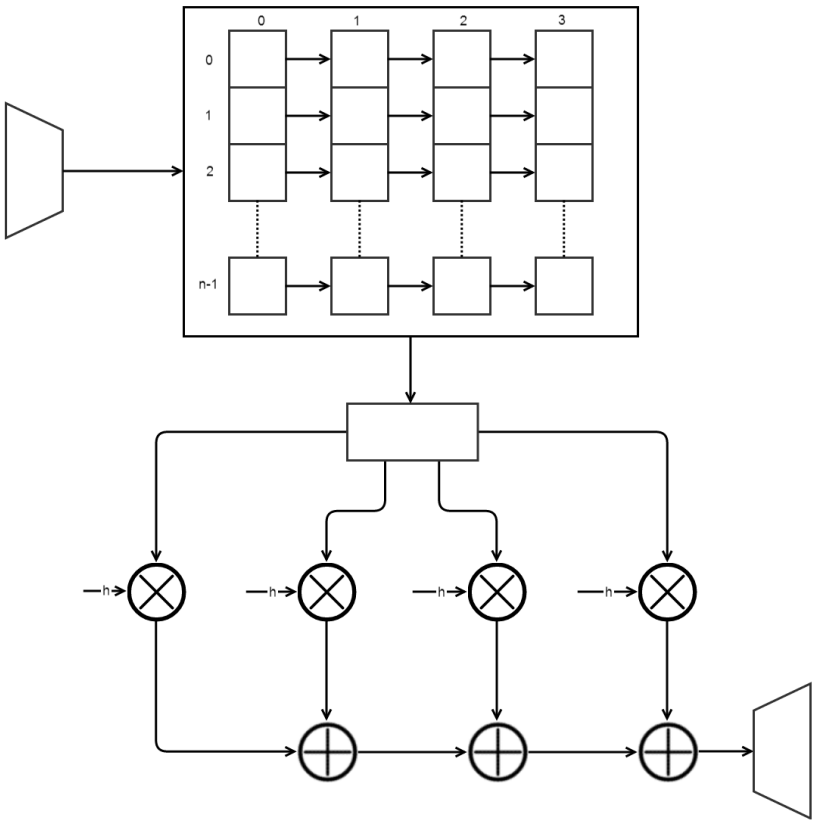

Figure 1: Architecture diagram of the system.

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 **??**.

```
reg [0:NR_STREAMS_LOG-1] stream;
```

We have 1024 streams. This register hold the index of the stream we are currently processing.

```
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 [0:L_LOG-1] m;
```

Holds a delayed value of l, this is required to compensate for the fact that I/O and computation happen simultaneously.

```
reg signed [0:DDWIDTH-1] in0 [0:NR_STREAMS-1];
reg signed [0:DDWIDTH-1] in1 [0:NR_STREAMS-1];
reg signed [0:DDWIDTH-1] in2 [0:NR_STREAMS-1];
reg signed [0:DDWIDTH-1] in3 [0:NR_STREAMS-1];
```

We need to store the 4 latest values of all 1024 input streams.

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

We need to store our coefficients. The coefficients are symmetric, so we can do an optimization storing only $2L + 1$ coefficients.

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

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

```
in0[stream] <= data_in;
in1[stream] <= in0[stream];
in2[stream] <= in1[stream];
in3[stream] <= in2[stream];
req_out_buf <= 0;
```

As seen in figure **??**, 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 $(l + M)$ mod 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 $L - M$.

```
sum <= in0[stream] * h[m] + in1[stream] * h[m+L] +
in2[stream] * h[L*2-m] + in3[stream] * h[L-m];
```

This is the filtering part, we have 4 multiplications and 3 additions.

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.

`1024x16-bit single-port RAM: 4`

Memory to store the latest 4 input values, for each of the 1024 streams. Each value is 16 bit.

`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
```

To calculate the expression $h[l + L]$.

```
11 bit adder:  1
```

To calculate the stream index. We have $2^{10} = 1024$ streams.

```
9 bit substractor:  2
```

To calculate $h[L * 2 - 1]$ and $h[L - l]$.
`8 bit 2-to-1 multiplexer`

To index the input registers.

```
1 bit register:  2
```

For `req_out_buf` and `req_in_buf`.

```
32 bit register:  1
```

To store the output sum.

```
10 bit register:  1
```

Used for to keep track of which stream we are processing.

```
8 bit comparator greater
```

To check for overflow of nM mod L.

3.2 Properties

ISE report the following timing statistics for our design with $n = 1024$.

- Synthesis report
  - Minimum period: 14.714ns
  - Maximum Frequency: 67.961MHz
- Post-PAR static timing report
  - Minimum period: 16.155ns
  - Maximum frequency: 61.900MHz

This leaves us with a maximum frequency that is lower than 100 MHz. But because we produce output every clock cycle, we can still process 1024 streams at once. Therefore the

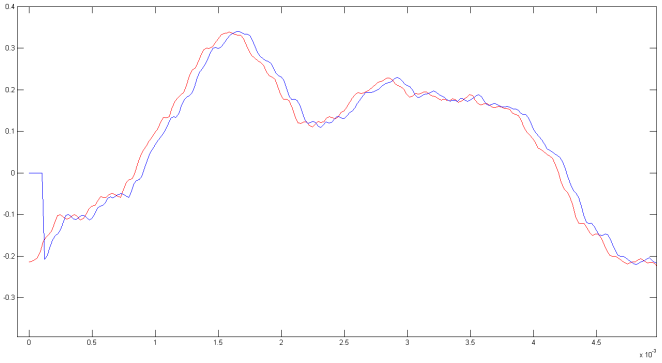

Figure 2: Plot of the first part of the original signal (red) and the signal after filtering (blue).

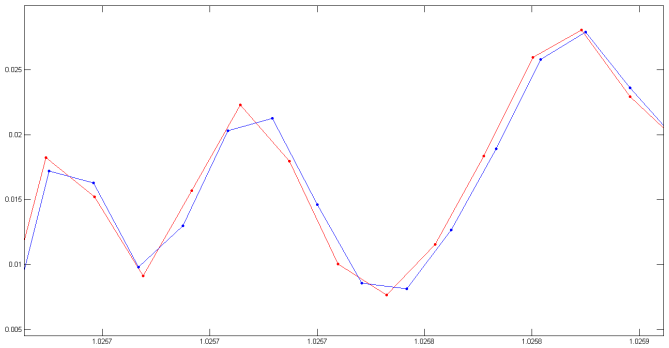

Figure 3: Plot of part of the original signal (red) and the signal after filtering (blue) with dots indicating the samples.

100 MHz requirement does not seem very important, it does however mean that we can not compile our design on the ngrid server.

3.3 Analysis of Filter Output

In this section we will show correctness of the upscaler by analyzing the input and output. Figure 2 shows the first part of the input and output signal. There is a finite amount of startup noise, indicated by the first part of the output signal being zero. It also shows the latency of the system, the output signal has some delay compared to the input signal. But except for those differences the signals appear identical. In figure 3 we see another plot of the input and output signal. In this plot we have shifted the output signal by -3 samples, and there are dots indicating the samples. We can now see that the output signal has a higher sample frequency than the input signal.

Figure 4 shows the signals in the frequency domain. There is only minimal difference between them. The output frequency contains a higher maximum frequency because it has a higher sample rate. Finally, figure 5 displays some waveforms of the design. For this

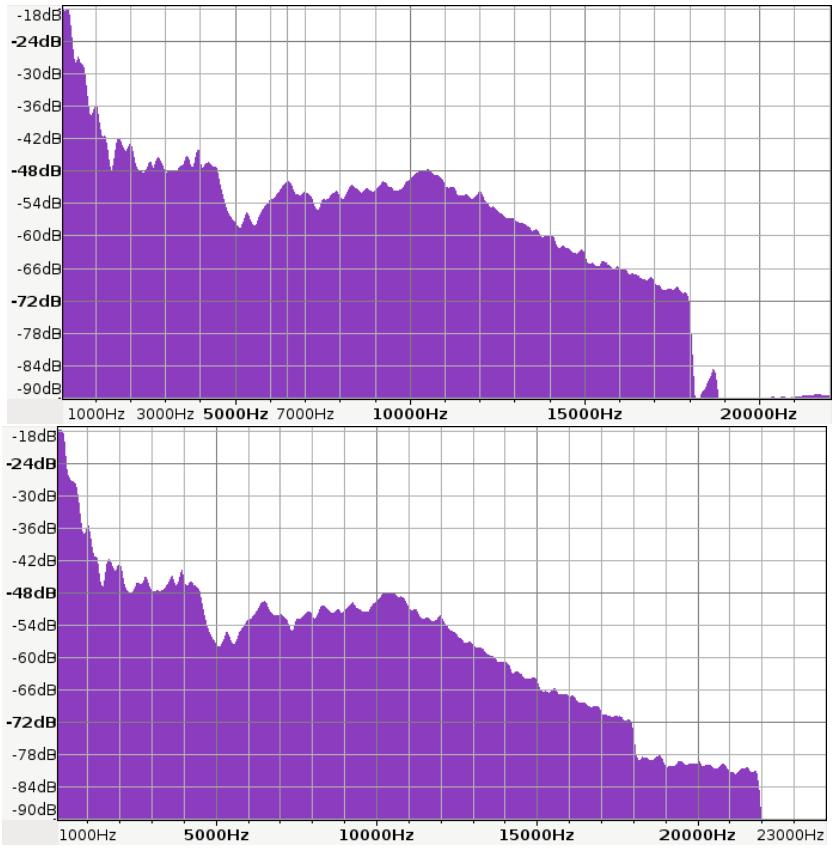

Figure 4: Plot in the frequency domain of the original signal (top) and the signal after filtering (bottom).

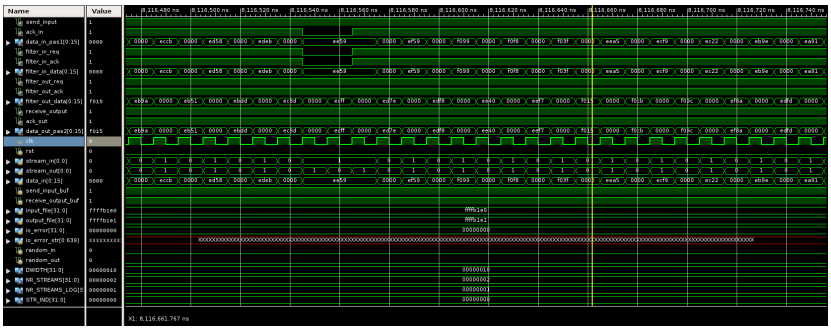

Figure 5: Part of the waveforms of our design.

image we have used $n = 2$. We can see this in the input and output streams, in which <0000> is interleaved with nonzero values. If we look at `clk` we can see that we do indeed produce output every clock cycle. We can also see a period in which `filter_in_ack` and `filter_req_ack` are zero, during which input and output remain stable.

4 Appendix A: Answers to inline questions

4.1 Question 1

We produce output every clock cycle. At 100 MHz we could produce $\frac{\frac{100 \cdot 10^3}{1}}{48} \approx 2083$ streams.

4.2 Question 2

Our design can process 1024 input streams. It does however not run at 100 MHz, we achieve this number of streams by producing output every clock cycle. The downside of this is that the `ngrid` server does not compile design not running at at least 100 MHz, so we could not test in on the Xilinx board.

4.3 Question 3

Section 3.3 contains an analysis of the input and output samples. There is one stream that represents an audio file, the others are zero. We have checked that the output audio is correct and has a sample frequency of 48 KHz. In our design we make no distinction between the streams, they are processed all in the same way. We do not use the number of the stream that contains actual data. Therefore it is reasonable to assume that all streams are correctly processed and have an output sample frequency of 48 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,
    parameter NR_STREAMS = 1024,
    parameter NR_STREAMS_LOG = 10)
(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;

reg [0:NR_STREAMS_LOG-1] stream;

// state counter. l = nM mod L (calculated efficiently using
// conditionals and addition/subtraction)
reg [0:L_LOG-1] l;
// Delayed state counter, to compensate for the fact that
// I/O and computation happen simultaneously now.
reg [0:L_LOG-1] m;
// The last 4 input samples used in the FIR (excluding
// the newest, which will be in data_in)
reg signed [0:DWIDTH-1] in0 [0:NR_STREAMS-1];
reg signed [0:DWIDTH-1] in1 [0:NR_STREAMS-1];
reg signed [0:DWIDTH-1] in2 [0:NR_STREAMS-1];
reg signed [0:DWIDTH-1] in3 [0:NR_STREAMS-1];

// 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];

// 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;
        stream <= 0;
        l <= 0;
    end
    // !Reset => run
    else
    begin

        // Read handshake complete
        if (req_in && ack_in)
        begin
            in0[stream] <= data_in;
            in1[stream] <= in0[stream];
            in2[stream] <= in1[stream];
            in3[stream] <= in2[stream];

            //sum <= (data_in >> 1) | (data_in & 32768);
            req_out_buf <= 1;
        end

           //Read handshake is pending then stop producing output
        if (req_in && !ack_in)
        begin
            req_out_buf <= 0;
        end

        // Write handshake complete
        if (req_out && ack_out)
        begin
            if( stream == 0 )
```

```
                begin
                    // l <= (l + M) mod L, implemented with conditionals
                    // No-overflow case
                    if( l < L - M )
                    begin
                        l <= l + M;
                        req_in_buf <= 0;
                    end
                    // Overflow case. Overflow also means we need a new input!
                    else
                    begin
                        l <= l - (L - M);
                        req_in_buf <= 1;
                    end

                    m <= l;
                end

                sum <= in0[stream] * h[m] + in1[stream] * h[m+L] + in2[stream]
                          * h[L*2-m] + in3[stream] * h[L-m];

                stream <= (stream + 1) & (NR_STREAMS-1);
            end

            //Write handshake is pending then stop acquiring output.
            if (req_out && !ack_out)
            begin
                req_in_buf <= 0;
            end

            // Idle state
            if (!req_in && !ack_in && !req_out && !ack_out)
            begin
                req_in_buf <= 1;
            end

        end
    end

endmodule
```