ECE 428 Final Project

Nick Hamann Matthew Morgan

May 4, 2009

Introduction

This report documents three approaches to designing and simulating a biquad low-pass filter. Structural Verilog is the design medium for the filter and Modelsim was used to produce raw output data while Matlab is used to plot and interpret the results. The output waveforms should indicate the attenuation of high frequencies and noise filtration.

First Design

The first design considered is a basic IIR filter with a critical delay path of one multipler and one adder. This design includes a simple data truncation at the output Y[N] such that the lower order bits are discarded completely. Refer to Figure 1(a) for the general block diagram. operands are eight bits, outputs of the multipliers are sixteen bits, and all adders are sixteen bit. The input and output waveforms, see Figure 1(b), indicate that this design is not optimal, however, we will see that further optimization also had little impact on this initial result.

First Verilog Code Iteration

Take note that no reset signal is needed. This reseting is handled by modeling a weak pull-down in the registers. In the event that the input value is unknown or high impedence, the output value defaults to zero. This helps to reduce the port list and also internal routing required for this control signal.

Listing 1: First Iteration

```
‘timescale 1ns / 1ps
module projfilt(x, clk, y);
input [7:0] x;
input clk;
output [7:0] y;
reg [7:0] y;

wire [15:0] d1, d2, d3;
reg [7:0] a1, b1, b2;
wire [15:0] m1, m2, m3, ad1, ad2;
wire s1, s2;

initial begin
        a1 = 8'hEE;
        b1 = 8'h51;
        b2 = 8'hDE;
        y = 8'h00;
end

mult8 a1mult(x, a1, m1);
mult8 b1mult(y, b1, m2);
mult8 b2mult(y, b2, m3);

reg16 reg1(m1, clk, d1);
reg16 reg2(ad1, clk, d2);
reg16 reg3(ad2, clk, d3);
```

```
add16 add1(d1, m3, 1'b0, ad1, s1);
add16 add2(d2, m2, 1'b0, ad2, s2);

// Dropped bit truncation
always @ (posedge clk)
begin
        y = d3[15:8];

end

endmodule
```

Truncation Code

The code used in the first design to truncate the lower order bits simply only sends the higher order eight bits to the output.

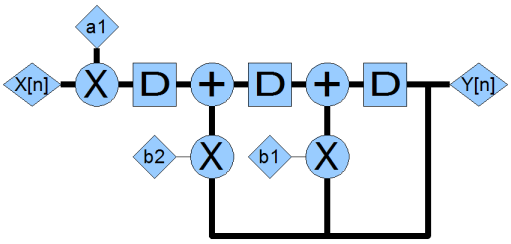
```
always @ (posedge clk)
begin
y = d3[15:8];
end
```

First Design Timing Analysis

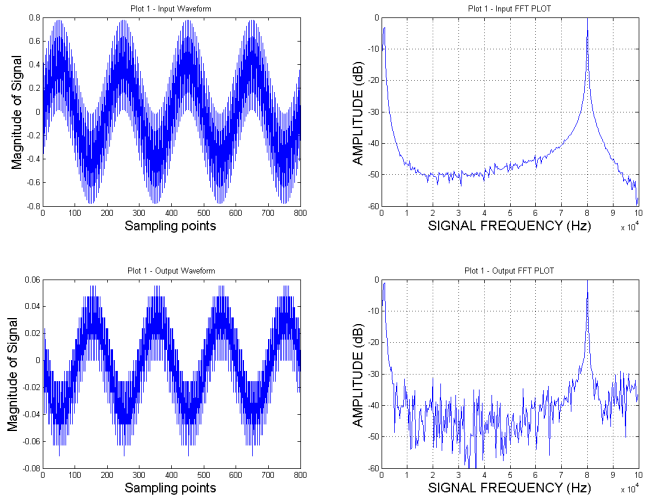
Table 1: Xilinx Timing Report

	Timing in (ns)
Delay	18.684
Requrement	100.000
Data Path Delay	18.538

According to Table 1, the maximum clock frequency for this design has been calculated to be approximately: 53.5 MHz.



(a) Functional block diagram



(b) Input/Output comparison

Figure 1: First Design Results

Second Design

The second design considered is a simple improvement over the first design which includes rounding before truncation a the output in an effort to improve resolution. This implementation will require another adder, though only eight bit instead of sixteen, that will increase the critical delay path from one multiplier and one adder to two adders and one multiplier. It should also be noted that this critical path will exist twice, refer to Figure 2(a). This increase in overall delay and probability of use is reflected in the timing analysis. Unfortunately, the output waveform shows little improvement in terms of attenuation; see Figure 2(b).

Rounding Truncation

The code used in the second design is uses an XOR gate to compare the ninth bit with the most significant or s̈ignb̈it. In the case that the number is a negative two's complement number, if the ninth bit is zero, the output is rounded up by one (add one), otherwise, the least significant eight bits are simply truncated as with the first design. This process is inverter in the case of positive numbers.

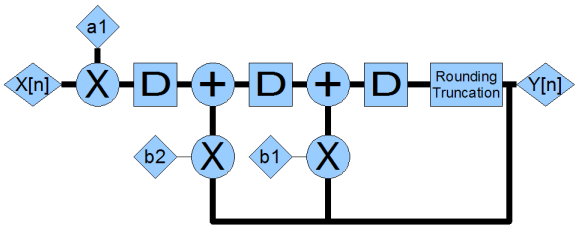
```
xor xor1(s3, d3[15], d3[7]);
add8 add3(d3[15:8], 7'b0, s3, 1'b0, ad3, cout3);
always @ (posedge clk) begin
y = ad3;
end
```

Second Design Timing Analysis

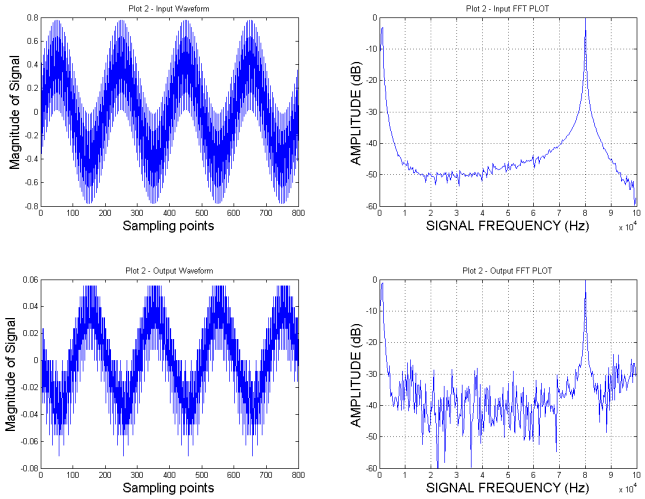
Table 2: Xilinx Timing Report

	Timing in (ns)
Delay	17.408
Requrement	100.000
Data Path Delay	17.408

According to Table 2, the maximum clock frequency for the second design has been calculated to be approximately: 57.4 MHz. This is a slight improvement over the first design. Though the critical path delay is technically longer, the eight bit rounding adder introduces a negligble delay and the simulation is largely unaffected.



(a) Functional block diagram



(b) Input/Output comparison

Figure 2: Second Design Results

Third Design

The third design introduces an effort to decrease the critical path but placing registers after the multipliers. The reset signal is re-introduced in this design iteration in order to eliminate any possiblitiy that this modification to the simulation program may be skewing the output; though it is believed that this is very unlikely.

Pipeling the Critical Path

The multiplier code has been altered in this design from the simple Verilog library multiplier to include the clocked storage of the output value. The removes the need to create a seperate module for a sixteen bit register. An eight bit register module is still required for the feed back after truncation.

‘timescale 100ns / 1ns
module mult8(a, b, p, clk, reset);
output signed [15:0] p;
input clk, reset;
input signed [7:0] a;
input signed [7:0] b;

reg signed [15:0] p;
wire signed [15:0] mult$_out$;

$assignmult_out = a * b;$

$always@(posedgeclk)$
$begin$
$if(reset == 1'b1)begin$
$p <= 16'b0;$
$endelsebegin$
$p <= mult_out;$
end
end

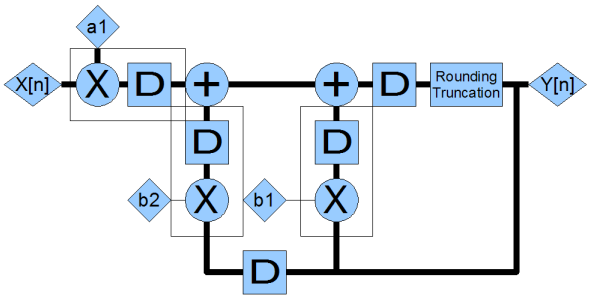
$endmodule$

Third Design Timing Analysis

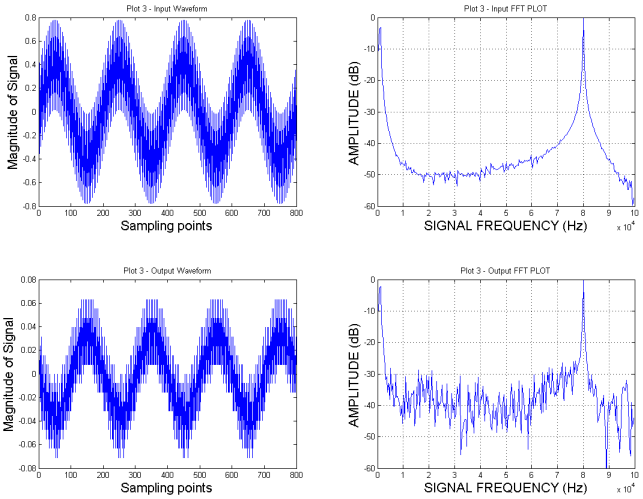
Table 3: Xilinx Timing Report

	Timing in (ns)
Delay	17.962
Requrement	100.000
Data Path Delay	17.951

According to Table 3, the maximum clock frequency for the this design has been calculated to be approximately: 55.7 MHz. This result is troubling as the goal of this iteration is to reduce the critical path delay and effectively increase the operating clock frequency. In fact, this design has increased the delay. After investigating, it seems that the ripple carry adders in use are the critical path as this design does not use a register between the two sixteen bit adders; refer to Figure 3(a). Again, the output waveform is largely ineffective in terms of filtering out high frequencies; refer to Figure **??** to review the results.



(a) Functional block diagram



(b) Input/Output comparison

Figure 3: Second Design Results

Fourth Design

Hand calculations reveal that due to the negative and positive constant operands to the multipliers, it is impossible to cause an underflow or overflow condition to occur. The previous ripple carry adder module handled overflow/underflow, however, in an effort to reduce adder delay, the optimized Verilog library adder is used in this iteration of design. It is hoped that this design will allow the pipelined multipliers to be leveraged.

Final Verilog Code Iteration

Due to time constraints the design process ends with this code for the filter.

Using Optimized Adders

module add$2_1 6(a, b, cin, sum, cout)$;
$outputreg[15:0]sum$;
$outputregcout$;
$input[15:0]a$;
$input[15:0]b$;
$inputcin$;

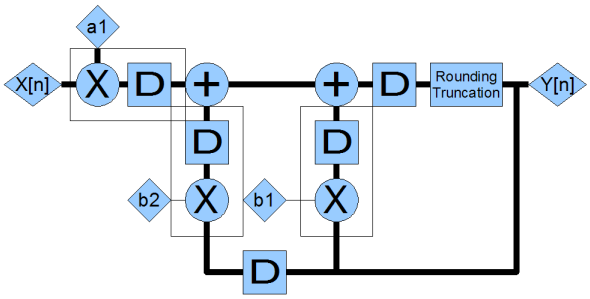
$always@(a, b, cin)$
$cout, sum = a + b + cin$;
$endmodule$

Fourth Design Timing Analysis

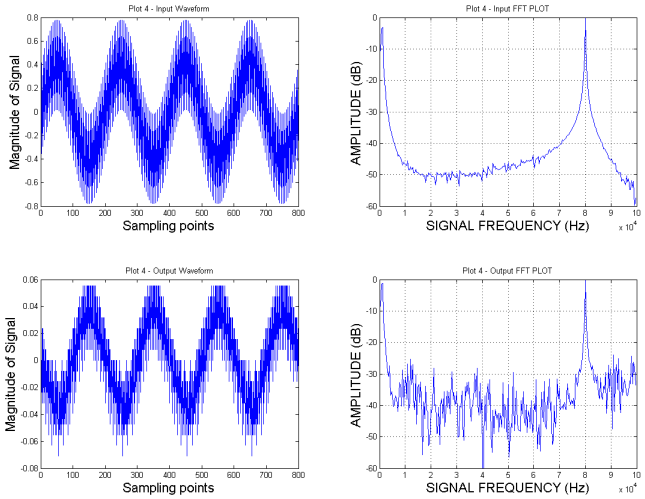
Table 4: Xilinx Timing Report

	Timing in (ns)
Delay	11.445
Requrement	100.000
Data Path Delay	11.369

According to Table 4, the maximum clock frequency for the this design has been calculated to be approximately: 87.4 MHz. This result proves that using the optimized library adder greatly reduces the critical path delay and increases clock frequency by over a third. Refer to Figure 4(b) to review the filter output. The block diagram for this design is the same as the third iteration, but is included for consistency.



(a) Functional block diagram



(b) Input/Output comparison

Figure 4: Second Design Results

Final Verilog Code Iteration

As a recap, changes that lead to this final design include: rounding before truncation (Listing 2), pipelined multipliers (Listing 5) to reduce path delay, and library based adders (Listing 3) to reduce the high delay incurred by the ripple carry. The Matlab code was modified to print both the input and output simultaneously to hasten the simulation and testing process. Refer to Listing 8 to review the changes to the Matlab code.

Listing 2: Final Filter Module

```
module projfilt(x, clk, reset, y);
input signed [7:0] x;
input clk, reset;
output signed [7:0] y;
reg signed [7:0] y;

wire signed [7:0] d1;
reg signed [7:0] a1, b1, b2;
wire signed [15:0] m1, m2, m3, ad1, ad2;
wire signed [7:0] ad3;
wire s1, cout1, cout2, cout3;

initial begin
        a1 = 8'hEE;
        b1 = 8'h51;
        b2 = 8'hDE;
        y = 8'h00;
end

mult8 a1mult(x, a1, m1, clk, reset);
mult8 b1mult(y, b1, m2, clk, reset);
reg8 reg1(y, clk, reset, d1);
mult8 b2mult(d1, b2, m3, clk, reset);


add2_16 add1(m1, m3, 1'b0, ad1, cout1);
add2_16 add2(ad1, m2, 1'b0, ad2, cout2);

xor xor1(s1, ad2[15], ad2[7]);
add2_8 add3(ad2[15:8], {7'b0, s1}, 1'b0, ad3, cout3);

// Dropped bit truncation
always @ (posedge clk)
begin
        if(reset == 1'b1)
```

```
                y <= 8'b0;
        else
                y <= ad3;
end

endmodule
```

Listing 3: Sixteen Bit Adder

```
module add2_16(a, b, cin, sum, cout);
  output reg [15:0] sum;
  output reg cout;
  input [15:0] a;
  input [15:0] b;
  input cin;

  always @(a, b, cin)
    {cout, sum} = a + b + cin;
endmodule
```

Listing 4: Eight Bit Truncation Adder

```
module add2_8(a, b, cin, sum, cout);
  output reg [7:0] sum;
  output reg cout;
  input [7:0] a;
  input [7:0] b;
  input cin;

  always @(a, b, cin)
    {cout, sum} = a + b + cin;
endmodule
```

Listing 5: Eight Bit Multiplier

```
module mult8(a, b, p, clk, reset);
output signed [15:0] p;
input   clk, reset;
input signed [7:0] a;
input signed [7:0] b;

reg signed [15:0] p;
wire signed     [15:0] mult_out;

assign mult_out = a * b;

always@(posedge clk)
begin
```

```
        if(reset == 1'b1) begin
                p <= 16'b0;
        end else begin
                p <= mult_out;
        end
end

endmodule
```

Listing 6: Eight Bit Register

```
module reg8(d, clk, reset, q);
input [7:0] d;
input clk, reset;
output [7:0] q;
reg [7:0] q;


always @ (posedge clk)
begin
        if(reset == 1'b1)
                q <= 8'b0;
        else
                q <= d;

end
endmodule
```

Listing 7: Matlab Code

```
`timescale 100ns / 1ns
module testfixture();

// Inputs
   reg [7:0] x;
   reg clk;
        reg reset;

// Output
   wire [7:0] y;

   integer fw;
   integer fr;
        integer fsc;
   integer i;

// Bidirs
```

```
// Instantiate the UUT
    projfilt UUT (
                    .x(x),
                    .clk(clk),
                    .reset(reset),
                    .y(y)
    );
// Initialize Inputs
       initial begin
                     x = 0;
                     clk = 0;
                     reset = 0;

   fr = $fopen("filter.in", "r");
   fw = $fopen("filter.out", "w");

   fsc = $fscanf(fr,"%b", x);
     #10 reset = 1;
     #10 clk = 1;
     #10 clk = 0;
     #10 reset = 0;


   for(i=0;i<800;i=i+1)
   begin
     #10 clk = 1;
     #10 clk = 0;
   end
   $fclose(fr);
   $fclose(fw);
   $stop;
end

always @(posedge clk)
begin
$fdisplay(fw, "%b", y);
fsc = $fscanf(fr,"%b", x);
end

endmodule
```

Listing 8: Firt Iteration

```
% read input signal
fin = get_result('filter.in', 8);
N = size(fin, 2);
```

```
t = 0:1:N-1;
% plot input signal
figure(1)
subplot(2,2,1)
plot(t, fin, 'b');
title('Plot 4 - Input Waveform');
ylabel('Magnitude of Signal', 'FontSize', 16)
xlabel('Sampling points', 'FontSize', 16)

% plot signal in frequency domain
%perform FFT
numpt = 512;
f1 = fin(1:numpt);
fclk=200000;
Dout_spect = fft(f1);
Dout_dB = 20*log10(abs(Dout_spect));
maxdB=max(Dout_dB(1:256));
figure(1)
subplot(2,2,2)
plot([0:numpt/2-1]*fclk/numpt,Dout_dB(1:numpt/2)-maxdB);
grid on;
title('Plot 4 - Input FFT PLOT');
xlabel('SIGNAL FREQUENCY (Hz)', 'FontSize', 16);
ylabel('AMPLITUDE (dB)', 'FontSize', 16);
hold off


% read output signal
fin = get_result('filter.out', 8);
N = size(fin, 2);
t = 0:1:N-1;
% plot output signal
figure(1)
subplot(2,2,3)
plot(t, fin, 'b');
title('Plot 4 - Output Waveform');
ylabel('Magnitude of Signal', 'FontSize', 16)
xlabel('Sampling points', 'FontSize', 16)

% plot output signal in frequency domain
%perform FFT
numpt = 512;
f1 = fin(1:numpt);
fclk=200000;
Dout_spect = fft(f1);
Dout_dB = 20*log10(abs(Dout_spect));
```

```
maxdB=max(Dout_dB(1:256));
figure(1)
subplot(2,2,4)
plot([0:numpt/2-1]*fclk/numpt,Dout_dB(1:numpt/2)-maxdB);
grid on;
title('Plot 4 - Output FFT PLOT');
xlabel('SIGNAL FREQUENCY (Hz)', 'FontSize', 16);
ylabel('AMPLITUDE (dB)', 'FontSize', 16);
axis([0, 10*10^4, -60, 0])
hold off
```

Concluding Remarks

Future design iterations may investigate a Multiplier Accumulator (MAC) implementation in order to simplify the number of modules instantiated. While the hand calculations coincide with simulations on the filter directly, it is possible that the original filter design is flawed and needs to be re-designed. Some unfound logical error may exist in the Verilog code which could be dropping samples and destroying the resolution. At this time, further investigation is required to resolve the strangely unaffected output.