ECE 428 Final Project

Nick Hamann Matthew Morgan

May 4, 2009

Design Objective

The objective of this project was to design and analyze a biquad low-pass digital filter. The filter is an IIR filter that can be represented by:

$$y[n] = -0.1372 * x[n-3] + 0.6249 * y[n-1] - 0.2667 * y[n-2] \tag{1}$$

The filter input, output and coefficients are 8-bit fixed-point two's complement numbers. All numbers are normalized so that each falls within the range of -1 to +1. The 8-bit two's complement representations of the filter coefficients are shown in Table 1.

Coefficient	Decimal	2's Complement Hex
a1	-0.1372	EE
b1	0.6249	51
b2	-0.2667	DE

Table 1: Filter Coefficient Representations

This report documents our approaches to designing the filter. The design was implemented in Verilog, with Model-Sim used to simulate the filter implementation and Xilinx used to analyze the performance of each filter design. Matlab was used to plot filter inputs and outputs.

Design Considerations

Two design issues that needed to be addressed in the filter design were data truncation and overflow/underflow handling.

Data Truncation

The first design issue considered was data truncation. Data truncation is necessary because when two 8-bit numbers (such as the input and a filter coefficient) are multiplied together, the result is a 15-bit number. The filter design needs to handle this in some way. Possible options include discarding extra bits or rounding after every multiplier so that the circuit never has to handle more than 8 bits. Another approach is to carry the full 15-bit output of each multiplier to the final output of the filter and perform discarding/rounding only at the end.

In order to increase the accuracy of the filter, it was decided to utilize the second approach. The circuit is small enough that the added cost of carrying the extra bits to the final ouput is negligible. Essentially, the only additional circuitry required is a modification of two adders to be 16-bit adders instead of 8-bit adders and possibly 16-bit registers instead of 8-bit registers depending on the design utilized.

Both discarding bits and rounding techniques were utilized in our design. We eventually settled on rounding because the obtained accuracy was desirable, especially given the negligible cost in both implementation and performance. The Design Details section discusses this in further detail.

Overflow/Underflow Handling

The second design issue considered was overflow/underflow handling. All of the values handled by the filter are normalized to fall within a range of -1 to +1. However, because the data width of the filter is fixed, it may result that a value is generated by the filter that exceeds +1 or is below -1.

Our initial design addressed this with additional behavioral logic in the Verilog adder circuit. If overflow was detected, the adder would output +1. If underflow was detected, the adder would output -1. It was later determined that this logic had little effect on the accuracy of the circuit, and was subsequently discarded in favor of performance gains from a more-optimized adder circuit. The Design Details section discusses this in further detail.

Design Details

First Approach

Description

The first filter approach utilized the block diagram shown in the project handout. This filter has three 8-bit multipliers, three 16-bit registers and two 16-bit adders. The design has a critical delay path of one multiplier and one adder. Truncation is performed at the end by simply discarding the 8 least-significant bits. A block diagram is shown in Figure 1. The full Verilog code can be found in the Verilog Code section.

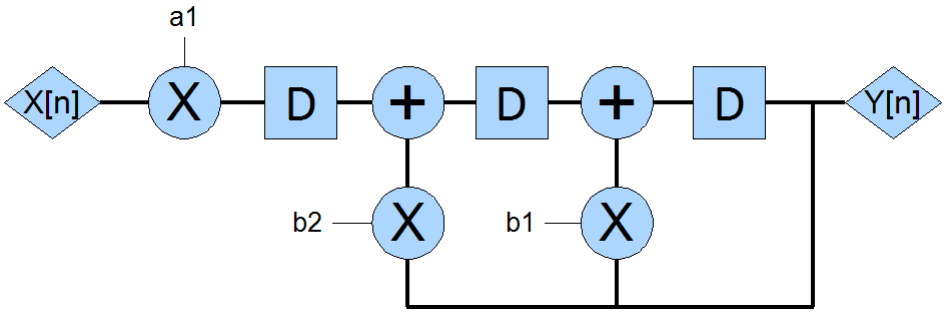

Figure 1: Block diagram of first filter design

This design does not utilize a reset signal. Instead, resetting is handled by modelling 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.

The overflow/underflow is handled by the logic found in the "always" block of add16.v. An if statement checks the XOR of the carry-in to the last bit with the carry-out of the last bit. If the result is 1, overflow or underflow has occurred. In order to determine which, the sign bits of the inputs must be checked. It is only necessary to check one of the inputs, since overflow or underflow can only occur when the inputs are either both positive or both negative. If the sign bit is found to be '0,' the output of the adder is +1. Otherwise the output is -1.

Listing 1: First approach overflow/underflow handling

```
always @(a or b or co or s) begin
        cout = co[14];
        if (co[14] ^ co[15])
                if(~b[15])
                        sum = {1'd0, 15'h7fff};
                else
                        sum = 16'h8001;
        else
                sum = s;
end
```

Performance

A Post-Place and Route Static Timing Analysis was performed on the design using Xilinx. The maximum delay path was found to be 17.408 ns, which correspondings to a maximum clock frequency of approximately 57.4 MHz. Details are shown in Table . The filter performance was simulated with a provided Verilog testbench file. A Matlab plot of the filter input and output can be found on attached pages.

Table 2: Xilinx Timing Report, First Design Approach

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

Discussion

The Matlab plots of the filter input/output reveal that little-to-no attenuation of the high-frequency component of the signal takes place. It was not understood why this was occurring, even after discussion with one of the TA's. We decided to implement rounding truncation instead of discard truncation in the next design iteration in the hopes of improving the filter's attenuation performance.

Second Approach

Description

The approach of the second design was to measure the performance advantages/disadvantages of performing rounding truncation over a simple discarding of the eight least-significant bits. Because the method of rounding used involves an 8-bit adder, performing rounding will increase the critical path delay to a mulitplier and two adders. It was desired to measure exactly this performance hit, as well as any possible increase in the accuracy of the filter. A block diagram is shown in Figure 2.

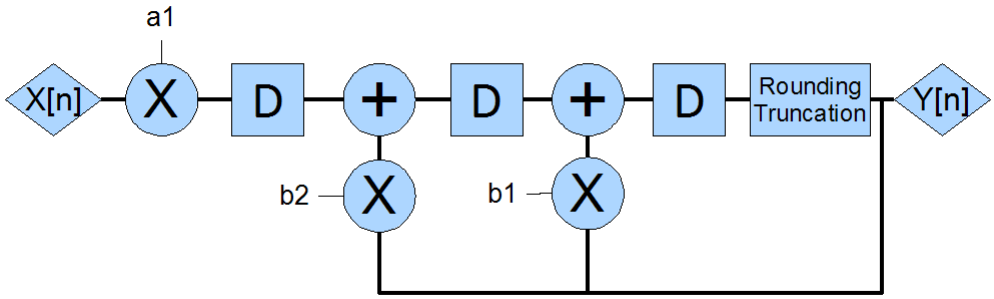

Figure 2: Block diagram of second filter design

The design is essentially the same, with the addition of add8.v, an 8-bit adder with exactly the same functionality as add16.v from the first design. This adder performs overflow/underflow handling in exactly the same way as previously described. In order to utilize this rounder, the following code was used to replace the "always" block of projfilt.v from the first design:

Listing 2: Round truncation code for second filter design

```
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
```

Performance

A Post-Place and Route Static Timing Analysis was performed on the design using Xilinx. The maximum delay path was found to be 18.684 ns, which correspondings to a maximum clock frequency of approximately 53.5 MHz. Details are shown in Table . The filter performance was simulated with a provided Verilog testbench file. A Matlab plot of the filter input and output can be found on attached pages.

Table 3: Xilinx Timing Report, Second Design Approach

	Time (ns)
Delay	18.684
Requrement	100.000
Data Path Delay	18.538

Discussion

The Matlab plots revealed that the rounding truncation had no effect on the attenuation performance of the filter, but did however slightly reduce the noise from the first design approach. This design approach did reveal, however, that the rounding truncation incurred only a slight speed penalty from simple discarding, so it was decided to keep this feature in subsequent designs.

Third Approach

The third design introduces an effort to decrease the critical path by placing registers after the multipliers. This so-called pipeline implementation utilizes three 16-bit registers, three 8-bit multipliers, two 16-bit adders, the same rounding truncation, but with an additional 8-bit register. A block diagram is shown in Figure 3. The Verilog code for this design approach is same as the code for the fourth approach. The fourth approach merely replaces the adders.

For this approach, the truncation and overflow/underflow handling techniques utilized are the same as in design two. The key difference is in a re-design of the multiplier code and the main filter organization. The

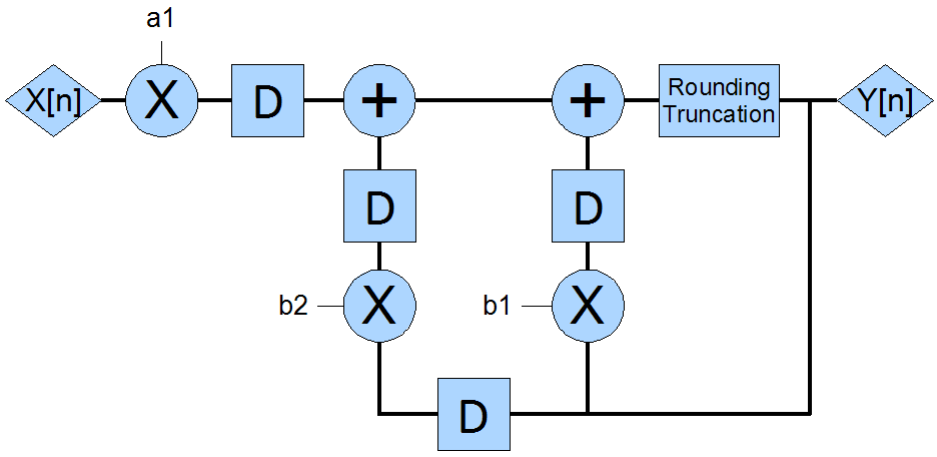

Figure 3: Block diagram of third/fourth filter design

mult8.v code now includes a register to store the output of the multiplier, which enables the pipeline filter implementation.

Performance

A Post-Place and Route Static Timing Analysis was performed on the design using Xilinx. The maximum delay path was found to be 17.962 ns, which correspondings to a maximum clock frequency of approximately 55.7 MHz. Details are shown in Table . The filter performance was simulated with a provided Verilog testbench file. A Matlab plot of the filter input and output can be found on attached pages.

Table 4: Xilinx Timing Report, Third Design Approach

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

Discussion

This result was peculiar because it demonstrated that our performance had actually *decreased* with the pipeline implementation. This should not have happened. The critical path delay for this third approach is three adders, while the critical path delay for approach 2 is one multiplier and two adders. This means that our adder was actually slower than the multiplier.

After considering the design, we determined that this was because our adder implementation was a basic structural carry-ripple adder. This likely caused the adder to be much slower than it should have been. In order to achieve any performance gain, a re-write of the adders was necessary.

Fourth Approach

Description

The final design approach we attempted utilized the same pipeline principle as well as the same truncation and overflow/underflow handling as in the third approach. The only difference was the migration away from the carry-ripple adders from design approaches 1 and 2 to optimized Verilog adders. The complete code listing for the fourth approach can be found in the Verilog Code section.

Performance

A Post-Place and Route Static Timing Analysis was performed on the design using Xilinx. The maximum delay path was found to be 11.445 ns, which correspondings to a maximum clock frequency of approximately 87.4 MHz. Details are shown in Table . The filter performance was simulated with a provided Verilog testbench file. A Matlab plot of the filter input and output can be found on attached pages.

Table 5: Xilinx Timing Report, Fourth Design Approach

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

Discussion

As expected, utilizing optimized adders greatly increased the performance of the circuit. The bottleneck was the adders. It should be noted here that the use of these library adders eliminated the overflow/underflow handling functionality. A look at the Matlab plots reveals, however, that this doesn't seem to have effected the attenuation performance of the filter at all. In fact, this design iteration has nearly exactly the same filter response as the last design iteration. The only change was a 57% increase in the speed of the circuit.

Conclusion

The obvious flaw with the discussed designs is that none of them seem to have achieved any sort of attenuation of high-frequency signal elements. In fact, a glance at the Matlab plots of each filter response reveal a stunning consistency in the inability to actually filter. We were not able to be determined what exactly was the root of this circumstance, but it is likely some fundamental error in the conception of the problem – a variety of multiplier and adder designs as well as truncation and overflow/underflow handling techniques seemed to have no effect on the high-frequency attenuation. We were able to improve the circuit performance primarily through use of optimized adders and a pipeline implementation, but the filter attenuation resisted such attempts.

Verilog Code

First Filter Design

Listing 3: projfilt.v

```
`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
```

Listing 4: mult8.v

```
`timescale 100ns / 1ns
```

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

assign p = a * b;

endmodule
```

Listing 5: reg8.v

```
`timescale 100ns / 1ns
module reg16(d, clk, q);
input [15:0] d;
input clk;
output [15:0] q;
reg [15:0] q;


always @ (posedge clk)
begin
if(d === 16'bz | d === 16'bx)
        q <= 16'b0;
else
        q <= d;

end

endmodule
```

Listing 6: add16.v

```
`timescale 1ns / 1ps
module add16(a, b, cin, sum, cout);
input [15:0] a, b;
input cin;
output [15:0] sum;
reg [15:0] sum;
output cout;
reg cout;

wire [15:0] s, co;

add1 a0(a[0], b[0], cin, s[0], co[0]);
add1 a1(a[1], b[1], co[0], s[1], co[1]);
add1 a2(a[2], b[2], co[1], s[2], co[2]);
add1 a3(a[3], b[3], co[2], s[3], co[3]);
add1 a4(a[4], b[4], co[3], s[4], co[4]);
```

```
add1 a5(a[5], b[5], co[4], s[5], co[5]);
add1 a6(a[6], b[6], co[5], s[6], co[6]);
add1 a7(a[7], b[7], co[6], s[7], co[7]);
add1 a8(a[8], b[8], co[7], s[8], co[8]);
add1 a9(a[9], b[9], co[8], s[9], co[9]);
add1 a10(a[10], b[10], co[9], s[10], co[10]);
add1 a11(a[11], b[11], co[10], s[11], co[11]);
add1 a12(a[12], b[12], co[11], s[12], co[12]);
add1 a13(a[13], b[13], co[12], s[13], co[13]);
add1 a14(a[14], b[14], co[13], s[14], co[14]);
add1 a15(a[15], b[15], co[14], s[15], co[15]);

always @(a or b or co or s) begin
        cout = co[14];

        if (co[14] ^ co[15])
                if(~b[15])
                        sum = {1'd0, 15'h7fff};
                else
                        sum = 16'h8001;
        else
                sum = s;
end

endmodule
```

Listing 7: add8.v

```
`timescale 100ns / 1ns
module add1(a, b, cin, sum, cout);
input a, b, cin;
output sum, cout;
reg sum, cout;

always @(a or b or cin) begin
        sum = (a ^ b) ^ cin;
        cout = (a & b) | (cin & (a ^ b));
end

endmodule
```

Fourth Filter Design

Listing 8: projfilt.v

```
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);

add16 add1(m1, m3, 1'b0, ad1, cout1);
add16 add2(ad1, m2, 1'b0, ad2, cout2);

// Rounding truncation
xor xor1(s1, ad2[15], ad2[7]);
add8 add3(ad2[15:8], {7'b0, s1}, 1'b0, ad3, cout3);

always @ (posedge clk)
begin
        if(reset == 1'b1)
                y <= 8'b0;
        else
                y <= ad3;
end

endmodule
```

Listing 9: mult8.v

```
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 10: reg8.v

```
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 11: add16.v

```
module add16(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 12: add8.v

```
module add8(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
```