Floating point for DSP on an Altera CycloneII FPGA – Bruce Land

Introduction

I teach a course at Cornell University in which students learn how to use Verilog and FPGAs to build processors and custom hardware. The goal is to build interesting devices such as robots, games, and lab instruments. What distinguishes the projects from general microcontroller projects is the high parallel throughput possible using parallel hardware on the FPGA. There is a definite need for a light weight floating point format in this class. Floating point arithmetic is very handy for designing filters and for other image and sound related computations. You can concentrate on the algorithm at hand without worrying about fixed-point scaling or overflow. Numerical dynamic range is hugely increased, although algorithm accuracy is always an issue. Also, numerical tools such as Matlab or Octave produce filter design results in floating format, so conversion of the design results to hardware is easier.

There are several dozen formats for floating point numbers (see Munafo) ranging from the high-accuracy 32-bit IEEE-754 standard to lowly 8-bit formats used in speech and video compression and for lecture examples. So why would I want to invent another format? There are several reasons. The basic reason follows from a quote from physicist Richard Feynman: “What I cannot create, I do not understand.” Building floating point hardware from scratch helps my understanding and teaching. Also, I was able to fit the algorithms more closely to the architecture of the CycloneII FPGAs we use at Cornell to teach ECE5760, Advanced microcontroller design. The close fit to the architecture makes it possible to instantiate up to 70 floating point multipliers on the CycloneII which ships with the Altera/Terasic educational DE2 prototype board. When we use the full IEEE floating point multiplier from Altera, we can fit only three multipliers on the FPGA we use. Finally, narrow floating point formats have been shown to be quite useful for DSP applications where IEEE-754 is overkill (Fang, et.al., Tong, et.al., Ehliar, et.al.).

To implement a floating point system, you need to pick a floating point representation and implement five basic operations necessary to use floating point for DSP and other fine-grained parallel operations. You need to be able to add, multiply, negate, and because audio and video codecs require fixed point integers, you need to convert integer-to-float and float-to-integer.

Floating point implementation

I decided to use 18-bit numbers because 18 bits is a native width for Altera’s M4K memory blocks and can be read or written in one clock cycle. Of those 18 bits, 9 are were used for the mantissa, one for the sign and 8 for the exponent. This format gives a numerical range of about $+/-10^{38}$. Any number smaller than about 10^{-38} underflows to zero. The resolution of the mantissa is only about

0.002, but this relatively low resolution is high enough for a range of DSP applications. Also, a 9-bit mantissa allowed me to use just one hardware multiplier. The mantissa is an unsigned fraction with the radix point just to the left of the top digit, so the maximum fraction is $1-2^{-9}$. I made the decision not to support denormalized fractions, so the minimum fraction is 0.5, with just the high-order bit of the mantissa set. If the number underflows, then the mantissa is set to zero. The sign bit is zero if the number is positive. The exponent is represented in 8-bit, offset binary, form. For example 2^0 is represented as 0h80, 2^2 as 0h82, and 2^{-1} as 0h7f. The Verilog representation for the 18-bit format is `{sign,exp[7:0],mantissa[8:0]}`. I did not implement the special numerical values available in IEEE-754 (NANs, infinities, denorms), so no bit patterns were allocated for these values. A few examples of decimal values and their floating format equivalents are shown below.

value	sign	exp	mantissa
0.5	0	0h80	0h100
-0.5	1	0h80	0h100
2.0	0	0h82	0h100
10.0	0	0h84	0h140
0.1	0	0h7d	0h199

Of the five operations which need to be implemented, negation is very easy, you just complement the sign bit. The other operations are more involved and are best understood as outlines. The first is multiplication:

1. If either input number has a mantissa high-order bit of zero, then that input is zero and the product is zero. This follows from the disallowing denorms.
2. If the sums of the input exponents is less than 128 then the exponent will underflow and the product is zero. This follows because the sum of exponents includes the 128 offset twice and therefore 128 must be subtracted from the input exponent sum.
3. If both inputs are nonzero and the exponents don't underflow then the product of the mantissas will be in the range from just less than one down to 0.25:

    1. If the simple product `(mantissa1)x(mantissa2)` has the high order-bit set (result>=0.5), then the top 9-bits of the product are the output mantissa and the output exponent is `exp1+exp2-128`.
    2. Otherwise the second bit of the product will be set (since the product of the mantissas must be greater than or equal to 0.25), and the output mantissa is the top 9-bits of the product shifted left one bit. The output exponent is `exp1+exp2-129` to account for the left shift of the mantissa.

4. The sign of the product is `(sign1)xor(sign2)`

Addition is actually a little more complicated than multiplication:

1. If both inputs are zero, the sum is zero.
2. Determine which input is bigger, which smaller (absolute value) by first comparing the exponents, then the mantissas if necessary.
3. Determine the difference in the exponents and shift the smaller input mantissa right by the difference.
    1. If the exponent difference is greater than 8 then just output the bigger input. The smaller number does not contribute significant bits.
    2. If the signs of the inputs are the same, add the bigger and (shifted) smaller mantissas. The result must be 0.5<sum<2.0. If the result is greater than one, shift the mantissa sum right one bit and increment the bigger input exponent, to become the output exponent. The sign is the sign of either input.
    3. If the signs of the inputs are different, subtract the bigger and (shifted) smaller mantissas so that the result is always positive. The result must be 0.0<difference<0.5. Shift the mantissa left until the high bit is set, while decrementing the bigger exponent once per shift, to become the output exponent. The sign is the sign of the bigger input.

It turns out that converting from integer to float is fairly simple. I assumed 10-bit, 2's complement, integers since the mantissa is only 9 bits, but the process generalizes to more bits.

1. Save the sign bit of the input and take the absolute value of the input.
2. Shift the input left until the high order bit is set and count the number of shifts required. This forms the floating mantissa.
3. Form the floating exponent by subtracting the number of shifts from step 2 from the constant 137 or (0h89-(#of shifts)).
4. Assemble the float from the sign, mantissa, and exponent.

Converting back to integer is similarly simple, but no overflow is detected, so scale carefully.

1. If the float exponent is less than 0h81, then the output is zero because the input is less than one.
2. Otherwise shift the floating mantissa to the right by (0h89-(floating exponent)) to form the absolute value of the output integer.
3. Form the 2's complement signed integer.

I coded the above outlines into Verilog for conversion to hardware on the FPGA. I wanted to see how fast I could make purely combinatorial floating point execute, so there is no pipelining or clocking of the arithmetic modules. Remember that every statement in Verilog represents the signal on a wire or bus and therefore every statement can change value simultaneously!

The code for the floating multiplier is shown below. The low level, unsigned, integer multiply of the mantissas is performed by a small module which gives the Altera QuartusII software a hint that a hardware multiplier should be used. The 9-bit by 9-bit multiply yields 18-bits of which 9 are selected for output in the asynchronous `always @(*)` statement. All of the modules are available for download from Circuit Cellar or from the course site (see references). The Altera QuartusII design software converted this multiplier code to about 60 logic elements plus one hardware multiplier on the CycloneII FPGA (out of 33,000 logic elements and 70 multipliers), while the adder takes about 220 logic elements. The timing analyzer suggests that the purely combinatorial multiplier should be able to run at 50 MHz and the adder at 30 MHz, and in fact run fine at 27 MHz.

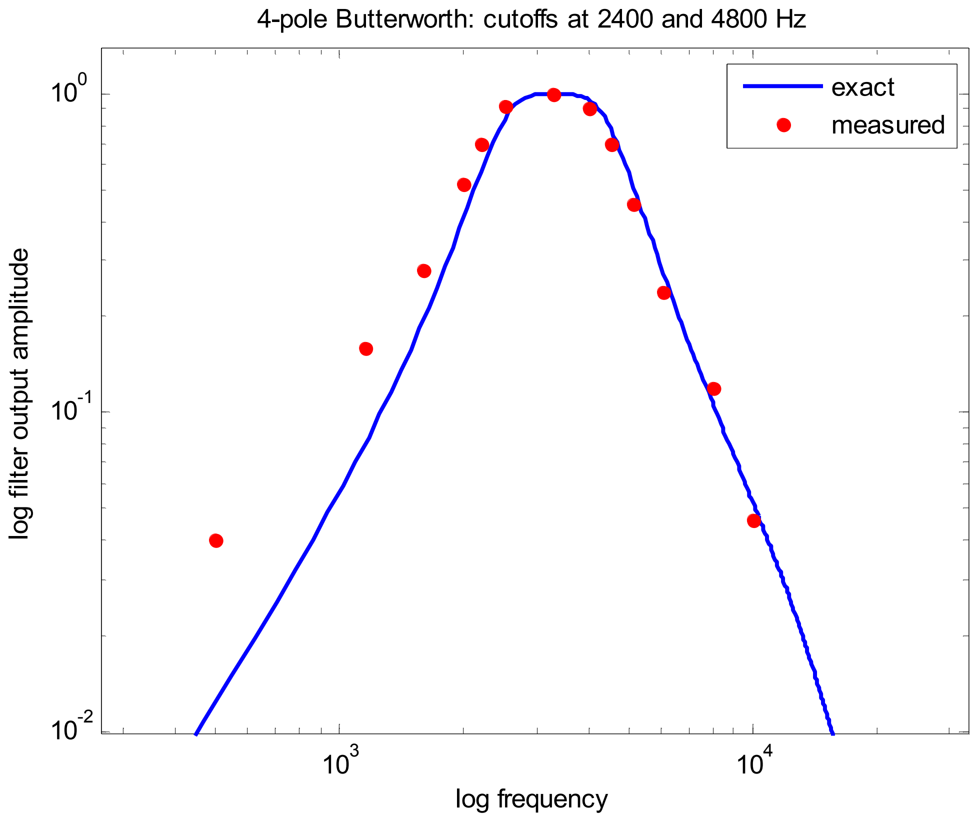
```
/////////////////////////////////////////////////////////
// floating point multiply
// -- sign bit -- 8-bit exponent -- 9-bit mantissa
// NO denorms, no flags, no NAN, no infinity, no rounding!
/////////////////////////////////////////////////////////
// f1 = {s1, e1, m1), f2 = {s2, e2, m2)
// If either is zero (zero MSB of mantissa) then output is zero
// If e1+e2<129 the result is zero (underflow)
//////////////////////////////////////////////////////////
module fpmult (fout, f1, f2);

	input [17:0] f1, f2 ; //the two floating inputs
	output [17:0] fout ; // the floating product

	wire [17:0] fout ;
	reg sout ;       // the output sign
	reg [8:0] mout ; // the output mantissa
	reg [8:0] eout ; // the output exponent extended to 9-bits for overflow

	wire s1, s2; // the two input signs
	wire [8:0] m1, m2 ; // the two input mantissas
	wire [8:0] e1, e2, sum_e1_e2 ; // extend to 9 bits to avoid overflow
	wire [17:0] mult_out ;    // raw multiplier output

	// parse f1
	assign s1 = f1[17];       // sign
	assign e1 = {1'b0, f1[16:9]};     // exponent extended one bit
	assign m1 = f1[8:0] ;     // mantissa
```

```
        // parse f2
        assign s2 = f2[17];
        assign e2 = {1'b0, f2[16:9]};
        assign m2 = f2[8:0] ;

        // first step in mult is to add extended exponents
        assign sum_e1_e2 = e1 + e2 ;

        // build output
        // raw integer multiply
        unsigned_mult mm(mult_out, m1, m2);

        // assemble output bits
        assign fout = {sout, eout[7:0], mout} ;

        always @(*)
        begin
                // if either is denormed or exponents are too small
                // the the output is zero
                if ((m1[8]==1'd0) || (m2[8]==1'd0) || (sum_e1_e2 < 9'h82))
                begin
                        mout = 0;
                        eout = 0;
                        sout = 0; // output sign
                end
                else // both inputs are nonzero and no exponent underflow
                begin
                        sout = s1 ^ s2 ; // output sign
                        if (mult_out[17]==1)
                        begin //MSB of product==1 normalized: result >=0.5
                                eout = sum_e1_e2 - 9'h80;
                                mout = mult_out[17:9] ;
                        end
                        else //MSB of product==0 result <0.5, so shift left
                        begin
                                eout = sum_e1_e2 - 9'h81;
                                mout = mult_out[16:8] ;
                        end
                end // nonzero mult logic
        end // always @(*)
endmodule
```

DSP application and testing.

To test the floating point modules I wrote a DSP application to filter an incoming audio signal through a 2, 4, or 6 pole, infinite impulse response filter. I figured that the actual audio input, plus the dynamics of the filters themselves would produce a large range of different floats. When the output of the filters had the correct frequency response and were free of artifacts, I could be reasonably sure that the modules were working correctly. At first, I tried to match the frequency response of the three filter types using a naïve, "Direct Form II Transposed" form similar to the Matlab filter function. It worked for the second order filters, but failed for the higher order filters because the 9-bit mantissa did not carry enough precision to represent the filter coefficients.

The solution was to factor the filters into second order sections (SOS). SOSs typically have coefficients which require lower accuracy, but more dynamic range, perfect for this floating point. Once rewritten as SOSs, the filter cutoff frequencies and phase shifts were close to the calculated values, implying that the floating point was working. The following figure shows a 4th order Butterworth response computed by Matlab in blue and the actual response of the FPGA implemented filter in red. The red points follow the exact solution fairly well, but diverge a little at low frequency.



It became tedious to use Matlab to generate the filter coefficients, then convert the coefficients into custom floating format, then write the Verilog, so I wrote a Matlab script to convert the filter specification to Verilog, given the order of the filter, the filter type (Butterworth, etc) and the cutoff frequencies. The script uses a Matlab signal processing toolbox function (`tf2sos`) to convert the filter to SOSs. The Matlab script output for a fourth order filter is shown below in Verilog source. The a's and b's and gain are the SOS filter constants.

```
//Filter: cutoff=0.100000
//Filter: cutoff=0.200000
IIR4sos filter4(
     .audio_out (filter4_out),
     .audio_in (audio_inR),
     .b11 (18'h10300),
     .b12 (18'h10500),
     .b13 (18'h10300),
     .a12 (18'h1037A),
     .a13 (18'h30185),
     .b21 (18'h10300),
     .b22 (18'h30500),
     .b23 (18'h10300),
     .a22 (18'h103BC),
     .a23 (18'h301B0),
     .gain(18'hF749),
     .state_clk(AUD_CTRL_CLK),
     .lr_clk(AUD_DACLRCK),
     .reset(reset)
) ; //end filter
```

The `IIr4sos` module is a state machine running at about 27 MHz which sequentially performs all the floating point filter operations in less than 2 microseconds, easily fast enough to keep up with a 48 kHz audio sample rate. The code is shown below and is summarized as follows:

1. Convert the 16-bit integer audio codec input to floating point.
2. Wait for the next 48 kHz audio clock edge to start the state machine
3. Compute the first floating point SOS as:
   y1(n) = b11*x(n) + b12*x(n-1) + b13*x(n-2) – a12*y1(n-1) – a13*y1(n-2)
   Where x(n) is the input at time n and y1(n) is the output at time n.
4. Update the filter state for the next time step
5. Compute the second floating point SOS as:
   y2(n) = b21*y1(n) + b22*y1(n-1) + b23*y1(n-2) – a22*y2(n-1) – a23*y2(n-2)
6. Update the filter state for the next time step
7. Multiply y2(n) by the gain input to form the filter output.
8. Convert the filter output back to 16-bit fixed point for the audio output codec.

In the code below, the floating point multiply-and-accumulate (MAC) operation sequentially takes its inputs from two registers `f_coeff` and `f_value` and places the result in `f_mac_new`. Most of the state machine consists of five MAC operations for each of the two SOS. State 15 stops the execution of the filter until the next audio sample becomes available. States 1 to 5 compute the MAC operations for SOS one, state 5 updates the history registers for SOS one and couples SOS one to SOS 2. State 8 to 12 compute the MAC operations for SOS two, state 13 updates the history registers for SOS two and couples SOS two to the output register.

```
///////////////////////////////////////////////////////////////////
/// Fourth order IIR filter -- written as two SOS ////////////////
///////////////////////////////////////////////////////////////////
module IIR4sos (audio_out, audio_in,
                       b11, b12, b13,
                       a12, a13,
                       b21, b22, b23,
                       a22, a23,
                       gain,
                       state_clk, lr_clk, reset) ;
//
// one audio sample, 16 bit, 2's complement
```

```
output wire signed [15:0] audio_out ;
// one audio sample, 16 bit, 2's complement
input wire signed [15:0] audio_in ;

// filter coefficients
input wire [17:0] b11, b12, b13, a12, a13, b21, b22, b23, a22, a23,
gain ;
input wire state_clk, lr_clk, reset ;

/// filter vars //////////////////////////////////////////////////
wire [17:0] f_mac_new, f_coeff_x_value ;
reg [17:0] f_coeff, f_mac_old, f_value ;

// input to the two SOS filters
reg [17:0] x1_n, x2_n ;
// input history x(n-1), x(n-2)
reg [17:0] x1_n1, x1_n2, x2_n1, x2_n2 ;

// output history: y_n is the new filter output, BUT it is
// immediately stored in f1_y_n1 for the next loop through
// the filter state machine
reg [17:0] f1_y_n1, f1_y_n2, f2_y_n1, f2_y_n2 ;

// i/o conversion
// int output of FP calc
wire [9:0] audio_out_int ;
reg [17:0] audio_out_FP ;
wire [17:0] audio_in_FP ;
int2fp f_input(audio_in_FP, audio_in[15:6], 0) ;
fp2int f_output(audio_out_int, audio_out_FP, 0) ;
assign audio_out = {audio_out_int, 6'h0} ;

// MAC operation
fpmult f_c_x_v (f_coeff_x_value, f_coeff, f_value);
fpadd f_mac_add (f_mac_new, f_mac_old, f_coeff_x_value) ;

// state variable
reg [3:0] state ;
//oneshot gen to sync to audio clock
reg last_clk ;
///////////////////////////////////////////////////////////////////

//Run the filter state machine FAST so that it completes in one
//audio cycle
always @ (posedge state_clk)
begin
        if (reset)
        begin
                state <= 4'd15 ; //turn off the state machine
        end

        else begin
                case (state)

                        1:
                        begin
                                // set up b11*x(n)
```

```
                                        f_mac_old <= 18'd0 ;
                                        f_coeff <= b11 ;
                                        f_value <= audio_in_FP ;
                                        //register input
                                        x1_n <= audio_in_FP ;
                                        // next state
                                        state <= 4'd2;
                                end

                                2:
                                begin
                                        // set up b12*x(n-1)
                                        f_mac_old <= f_mac_new ;
                                        f_coeff <= b12 ;
                                        f_value <= x1_n1 ;
                                        // next state
                                        state <= 4'd3;
                                end

                                3:
                                begin
                                        // set up b13*x(n-2)
                                        f_mac_old <= f_mac_new ;
                                        f_coeff <= b13 ;
                                        f_value <= x1_n2 ;
                                        // next state
                                        state <= 4'd4;
                                end

                                4:
                                begin
                                        // set up a12*y(n-1)
                                        f_mac_old <= f_mac_new ;
                                        f_coeff <= a12 ;
                                        f_value <= f1_y_n1 ;
                                        // next state
                                        state <= 4'd5;
                                end

                                5:
                                begin
                                        // set up a13*y(n-2)
                                        f_mac_old <= f_mac_new ;
                                        f_coeff <= a13 ;
                                        f_value <= f1_y_n2 ;
                                        // next state
                                        state <= 4'd6;
                                end

                                6:
                                begin
                                        // get the output of the first SOS
                                        // and put it in the LAST output var
                                        // for the next pass thru the state
machine
                                        f1_y_n1 <= f_mac_new ;
                                        // link first SOS to second SOS
```

```
        x2_n <= f_mac_new ;
        // update output history
        f1_y_n2 <= f1_y_n1 ;
        // update input history
        x1_n1 <= x1_n ;
        x1_n2 <= x1_n1 ;
        //next state
        state <= 4'd8;
end

8:
begin
        // set up b21*x(n)
        f_mac_old <= 18'd0 ;
        f_coeff <= b21 ;
        f_value <= x2_n ;
        // next state
        state <= 4'd9;
end

9:
begin
        // set up b22*x(n-1)
        f_mac_old <= f_mac_new ;
        f_coeff <= b22 ;
        f_value <= x2_n1 ;
        // next state
        state <= 4'd10;
end

10:
begin
        // set up b23*x(n-2)
        f_mac_old <= f_mac_new ;
        f_coeff <= b23 ;
        f_value <= x2_n2 ;
        // next state
        state <= 4'd11;
end

11:
begin
        // set up a22*y(n-1)
        f_mac_old <= f_mac_new ;
        f_coeff <= a22 ;
        f_value <= f2_y_n1 ;
        // next state
        state <= 4'd12;
end

12:
begin
        // set up a23*y(n-2)
        f_mac_old <= f_mac_new ;
        f_coeff <= a23 ;
        f_value <= f2_y_n2 ;
        // next state
```

```
                                        state <= 4'd13;
                                end

                                13:
                                begin
                                        // get the output
                                        // and put it in the LAST output var
                                        // for the next pass thru the state
machine
                                        f2_y_n1 <= f_mac_new ;
                                        // apply the final gain mult
                                        f_value <= f_mac_new ;
                                        f_coeff <= gain ;
                                        // update output history
                                        f2_y_n2 <= f2_y_n1 ;
                                        // update input history
                                        x2_n1 <= x2_n ;
                                        x2_n2 <= x2_n1 ;
                                        //next state
                                        state <= 4'd14;
                                end

                                14:
                                begin
                                        audio_out_FP <= f_coeff_x_value ;
                                        //next state
                                        state <= 4'd15;
                                end

                                15:
                                begin
                                        // wait for the audio clock and one-shot
it
                                        if (lr_clk && last_clk==1)
                                        begin
                                                state <= 4'd1 ;
                                                last_clk <= 1'h0 ;
                                        end
                                        // reset the one-shot memory
                                        else if (~lr_clk && last_clk==0)
                                        begin
                                                last_clk <= 1'h1 ;
                                        end
                                end

                                default:
                                begin
                                        // default state is end state
                                        state <= 4'd15 ;
                                end
                        endcase
                end
end

endmodule
```

Conclusions

The 18-bit floating point described here is allows up to 70 floating point multipliers and around 150 floating point adders to be placed on the 33,000 logic element CycloneII FPGA which is standard on the the Altera DE2 educational development board. At a 30 MHz clock rate this would allow around 6 billion floating point operations/second, enough for serious audio processing and even some video processing.

One student project from last semester used the floating point routines to implement a polygon rendering pipeline on the FPGA (Penmetcha and Pryor). The pipeline worked, another good test for the floating point routines, although 9 bit resolution on the mantissa is a little low for good z-buffering.

References:

Fang Fang, Tsuhan Chen, Rob A. Rutenbar, *Lightweight floating-point arithmetic: Case study of inverse discrete cosine transform*, EURASIP J. Sig. Proc.; Special Issue on Applied Implementation of DSP and Communication Systems(2002)

Fang Fang, Tsuhan Chen, and Rob A. Rutenbar, *FLOATING-POINT BIT-WIDTH OPTIMIZATION FOR LOW-POWER SIGNAL PROCESSING APPLICATIONS*, http://amp.ece.cmu.edu/Publication/Fang/icassp02_Fang.pdf (2002)

Jonathan Ying Fai Tong, David Nagle, Rob. A. Rutenbar, *Reducing power by optimizing the necessary precision/range of floating-point arithmetic*, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Volume 8 , Issue 3 (2000) Special section on low-power electronics and design Pages: 273 - 285

Eilert, J. Ehliar, A. Dake Liu, *Using low precision floating point numbers to reduce memory cost for MP3 decoding*, IEEE 6th Workshop on Multimedia Signal Processing, 2004 : 29 Sept.-1 Oct. 2004 page(s): 119- 122
http://www.da.isy.liu.se/pubs/eilert/eilert-mmsp2004.pdf

Robert Munafo, Survey of Floating-Point Formats,
http://www.mrob.com/pub/math/floatformats.html

Altera educational program: for QuartusII download and information on hardware.
http://www.altera.com/education/univ/unv-index.html

ECE 5760 main page
http://instruct1.cit.cornell.edu/courses/ece576/

ECE 5760 Floating point page
http://instruct1.cit.cornell.edu/courses/ece576/FloatingPoint/index.html

Penmetcha and Pryor, ECE 5760: Graphics Processing Unit,
http://instruct1.cit.cornell.edu/courses/ece576/FinalProjects/f2008/ap328_sjp45/website/introduction.htm

Variable width floating point in Verilog and VHDL
http://www.eda.org/fphdl/