Homework 8: Due Friday 11/5 2:10 pm.

In this homework set, we’ll work with an 8-bit versions of IEEE 754 Floating Point – CS150 Floating Point, as shown below:

<table>
<thead>
<tr>
<th></th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sign Bit</td>
<td>Exponent</td>
<td>Significand</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CS150 FP has 1 sign bit, 3 exponent bits, and 4 significand bits. It uses the normalized version with the hidden 1 for the significand. The exponent is in excess 3.

**Problem 1.** To get practice with floating point operations, complete the following worksheet by hand. For the arithmetic problems, you should round your result toward zero (truncate). Show your work and note any overflow that occur.

<table>
<thead>
<tr>
<th>Convert From Floating Point to Decimal</th>
<th>0 010 1110</th>
<th>0 110 0111</th>
<th>1 100 1001</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.9375</td>
<td>11.5</td>
<td>-3.125</td>
</tr>
</tbody>
</table>

**Floating Point Addition** (Write both FP and what your FP is in decimal)

<table>
<thead>
<tr>
<th></th>
<th>0 010 1110</th>
<th>0 010 1110</th>
<th>0 110 0111</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+ 0 110 0111</td>
<td>+ 1 100 1001</td>
<td>+ 1 100 1001</td>
</tr>
<tr>
<td></td>
<td>0 110 1000</td>
<td>1 100 0001</td>
<td>0 110 0000</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>2.125</td>
<td>8</td>
</tr>
</tbody>
</table>

**Floating Point Multiplication** (Write both FP and what your FP is in decimal)

<table>
<thead>
<tr>
<th></th>
<th>0 010 1110</th>
<th>0 010 1110</th>
<th>0 110 0111</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x 0 110 0111</td>
<td>x 1 100 1001</td>
<td>x 1 100 1001</td>
</tr>
<tr>
<td></td>
<td>0 110 0101</td>
<td>1 100 0111</td>
<td>OVERFLOW!</td>
</tr>
</tbody>
</table>
Problem 2. In this problem you are going to construct the data path for a cs150 floating point adder. Your adder will need to adjust the normalized numbers so that they can be added, add the numbers and return a normalized solution. There are no denorms, NANs, or Infinities. Just the basics. Round toward zero. Draw the data path for this adder.

Write your FP adder in verilog.

module fpadd(a, b, result);

    input [7:0] a; // the 1st single-precision FP operand
    input [7:0] b; // the 2nd FP operand
    output [7:0] result; // the final, correctly rounded sum of the 2 operands

    wire [5:0] x; // aligned mantissa of larger operand
    wire [11:0] y; // aligned mantissa of smaller operand
    wire [2:0] biggerexp; // the value of the larger exponent
    wire [6:0] sum; // the un-normalized sum after mantissa addition
    wire [3:0] normsum; // the normalized sum
    wire [3:0] normalshift; // the amount the un-normalized sum was shifted to get normalized sum
    wire presticky; // or of all bits discarded during mantissa addition
    wire guard; // guard bit
    wire [126x516]effop; // the effective operation (operation to be performed on mantissae)
    wire [3:0] inex; // inexact after rounding
    wire [3:0] overexp; // exponent with shifts and biases
    wire [3:0] round; // round bit
    wire sticky; // sticky bit
    wire zero; // high if final result is 0
    wire effop; // the effective operation (operation to be performed on mantissae)
    wire finalsig; // final sign of FP result
    wire [3:0] roundsum; // the final sum rounded and without leading one including post-normalization
    wire [3:0] roundshift; // the total amount the sum has been shifted
    wire [3:0] exp; // somewhat final exponent

    fpalign fpalign(a[6:0], b[6:0], x, y, biggerexp, abig);
    mantadd mantadd(a[7], b[7], x, y, op, sum, presticky, guard, effop);
    normlize normlize(sum, biggerexp, presticky, guard, effop, normsum, normalshift, round, sticky, zero, inex, overexp);
module fpalign(a, b, x, y, biggerexp, abig);

input [6:0] a;       // the 1st single precision FP input
input [6:0] b;       // the 2nd FP input
output [5:0] x;      // the aligned mantissa of bigger
output [11:0] y;     // the aligned mantissa of smaller
output [2:0] biggerexp; // the bigger exponent
output [2:0] abig;   // high if expa bigger than expb
wire [2:0] expa;     // exponent of a
wire [2:0] expb;     // exponent of b
wire [2:0] smallerexp; // the smaller of expa and expb
wire [2:0] shift;    // amount to shift smaller mantissa
wire azero;         // is a or b 0 or denorm?
wire bzero;
wire [3:0] shiftamount; // final amount to shift for alignment
wire smallshift;    // high if shift is smaller than max necessary
wire [5:0] aval;     // mantissa of a
wire [5:0] bval;     // mantissa of b
wire [12:0] yprelim; // y before alignment shift

assign azero = ~|a[6:4];       // logic to see if expa or expb is 0
assign bzero = ~|b[6:4];

// if expa or expb is 0, it is set to 1 for denorm handling
assign expa = azero ? 3'b0 : a[6:4]; // put the exponent of a into expa
assign expb = bzero ? 3'b1 : b[6:4]; // put exponent of b into expb
assign abig = (a[6:0] > b[6:0]); // is expa bigger than expb
assign biggerexp = abig ? expa : expb;       // save whichever exponent is bigger
assign smallerexp = abig ? expb : expa;       // store smaller exponent

// to get shift amount, smaller subtracted from larger
assign shift = biggerexp - smallerexp;
// determines final amount to shift, never have to shift more than 6
assign smallshift = shift < (7);
assign shiftamount = smallshift ? shift[3:0] : 7;
assign aval = { ~azero, a[3:0], 1'b0};
assign bval = { ~bzero, b[3:0], 1'b0};
assign yprelim = { (abig ? bval : aval), 6'b0};
module mantadd(sa, sb, x, y, sum, presticky, guard, effop);

input sa;              // sign of 1st FP operand
input sb;              // sign of 2nd FP operand
input [5:0] x;         // the bigger mantissa
input [11:0] y;        // the smaller mantissa
output [6:0] sum;      // the result after the desired operation is performed on mantissae
output presticky;     // keeps track of or of discarded bits
output guard;         // bit that will be shifted into mantissa if MSB cancels on sub
wire [7:0] yinput;     // the version of y put into the adder (either y or ~y)
wire carry;           // the carry input to the adder

assign effop = sa ^ sb;         // this is the effective operation (add=0; sub=1)
assign presticky = |y[4:0];      // if any discarded bits 1, this is high
assign guard = y[5];            // guard bit is just beyond cutoff point
assign yinput = effop ? ~{1'b0, y[11:6]} :y[11:6];    // choose between y for addition or ~y for subtraction
assign carry = ~(presticky | guard) & effop; // sometimes have to add one for 2's complement
assign sum = x + yinput + carry; // computing sum (25 bit adder with additional carry in)
endmodule

module normalize(sum, biggerexp, presticky, guard, effop, normsum, normalshift, round, sticky, zero, inex, overexp);

input [6:0] sum;            // the un-normalized sum
input [2:0] biggerexp;      // the larger of expa and expb
input presticky;            // sticky bit
input guard;                // guard bit, shifted into significand if MSB cancels
input effop;                // effective operation
output [3:0] normsum;       // the normalized sum
output [3:0] normalshift; // this will hold the amount to shift sum for normalization
output round; // the rounding bit
output sticky; // sticky bit for rounding
output zero; // whether or not final mantissa is 0
output denorm; // high if result is denormal
output inex; // if there are round or sticky bits, the final result will be inexact

output [3:0] overexp; // exponent with all shifts and biases taken into account
wire [2:0] biasexp; // exponent with trapped under/overflow bias
wire [3:0] shiftamount; // amount to left shift sum
wire [6:0] shiftedsum; // holds sum shifted by normalization amount
wire shifttwo; // high if MSB cancels on subtraction

assign overexp = biggerexp - normalshift;

// chooses whether to use denorm shifting or leading zero shifting
assign shiftamount = (overexp[3]) ? biggerexp : normalshift;

// shift the sum the amount determined by number of leading 0's
// or by biggerexp+1 for denormalized numbers
assign shiftedsum = sum << shiftamount;

// determines whether the MSB will cancel when subtracting

// these break the shifted sum into its components: the 23 bits that will actually become the new FP # and the
// round and sticky bits, which will be used for rounding
assign normsum = shiftedsum[5:2];
assign round = shifttwo ? (presticky ^ guard) : (~|normalshift[3:1] & effop) & shiftedsum[1];
assign sticky = shifttwo ? presticky : (shiftedsum[0] | guard | presticky);
assign zero = ~|shiftedsum; // tests to see if result is zero

// logic to determine if result is denormalized
assign denorm = overexp[3] & |normsum;
assign inex = round | sticky; // if some non-0 bits were cut off, the result will be inexact

endmodule

module rounder(normsum, round, sticky, roundmode,
finalsign, overexp, roundsum, roundshift, rm, rz, rp, rn, exp);

input [3:0] normsum; // the normalized sum
input round; // round bit
input sticky; // the sticky bit
input [1:0] roundmode; // round mode selection
input finalsign; // final sign of FP result from final
input [2:0] overexp; // the exponent calculated in normlize with shifting
output [3:0] roundsum; // the correctly rounded sum in significand form (no
leading 1)
output roundshift; // indicates whether there was overflow during
mantissa addition
output [3:0] exp; // the final, non special-case exponent

wire addone; // high if will have to add 1 during
rrouanding
wire [4:0] overflowsum; // sum with extra bit for overflow
wire [1:0] overshift; // correction amount, 2 if overflow during
round

// result in the event of rounding up
assign overflowsum = normsum + 1;

// determines if the exponent overflows for use in overflow determination
assign roundshift = overflowsum[4] & addone;

// this changes the normalized mantissa to what is dictated by the rounding mode
assign roundsum = addone ? overflowsum[3:0] : normsum;
assign overshift = roundshift ? 2 : 1; // overshift logic
assign exp = overexp + overshift; // formula to calculate final exponent

daendmodule

module final(a, b, sa, sb, abig, expneg, exp, effop, zero, roundsum, result);

input [2:0] a; // operand a
input [2:0] b; // operand b
input sa; // sign of a
input sb; // sign if b
input abig; // a > b ?
input expneg; // normalshift > biggerexp
input [3:0] exp; // calculated final exponent
input effop; // effective operation
input zero; // result is 0
input [3:0] roundsum; // the calculated final significand
output [7:0] result;  // resulting FP number
wire [2:0] biasexp;  // exponent with over/underflow bias
wire preover;  // pre-overflow calculation
wire [2:0] finalexp;  // the truly final exponent
wire [3:0] finalmant;  // the truly final significand
wire finalsign;  // the final sign of the result

// these wires are all just mux results
wire signmux;

assign signmux = (zero ? (sa & sb & ~op) : (abig & sa) | ((~abig | sa)));
assign finalexp = exp[2:0];
assign finalmant = roundsum;
assign result = {finalsign, finalexp, finalmant};  // putting the pieces together

endmodule

In words, how would you change the adder if you need to check for infinity?

You need flags to see if the exponent has become too large.
**Problem 3. Linear Feedback Shift Register as a Random Number Generator**

This problem is a more open-ended design exercise where you will need to produce a design from a description. You will design a subsystem that provides exponential random backoff support for transmission on a shared communication medium, such as might be used with Ethernet, wifi, or 802.15.4. The transmitter is a black-box that your subsystem will interface to. When the transmitter starts a backoff sequence it asserts a signal “start” telling your subsystem to produce a random wait-time over the minimum wait interval (minWait). Your subsystem will produce an (n-bit) result that represents an unsigned number specifying the number of clock cycles that the transmitter should delay before attempting to transmit. It will assert a signal, “done”, when this output is valid. After the wait, the transmitter determines whether the channel is busy. If not, it will transmit. If so, it will request a random wait time over an exponentially larger interval (twice the length) up to some maximum wait interval (maxWait). So the handshake is Start | Done | … | Next | Done | … | Next | Done, until the transmission succeeds. Each wait is a random length. The intervals that the waits are drawn from increase, up to a maximum value.

The clock frequency is 24 MHz, minWait is ~0.042 ms and maxWait is ~171 ms.

4104000 clock cycles max wait  
1,008 clock cycles min wait

You will use an LFSR to generate the random numbers. (It can just run continuously.)

a) How many bits wide must the output be to specify the maximum wait time? 
22 bits will be necessary.

b) How many of these will be used (i.e., potentially non-zero) for the minimum wait time? 
10 bits will be necessary.

(c) How can you extract a random number over the maximum interval? The Minimum interval? Mask the random number with 22 1’s for the maximum interval. Mask the random number with 10 1’s for the minimum interval.

d) Draw the schematic for your design using flipflops, registers, xor gate, shift registers, combinational logic. Show the control and data lines that form the inputs and output of your subsystem. (Hint: use a shift register to generate a mask of the random number.)
22 Bit Shift Register

Q22
CLK

QD
... 
Q4
Q3
Q2
Q1

QD

QD

QD

QD

clr
shift
**Problem 4.** In class we discussed how to implement SECDED. For example, with 8 information bits (d1-d8) we have 4 SEC parity bits (p1-p4) and one DED parity bit p. Before writing a word, we compute each Pi over its parity group. Then we compute p over the Di’s and Pi’s. Then we store all the Di, Pi, and P. On reading, we compute the parity over each group (an SEC parity bit and the information bits that it covers) to produce the check bits, c1-c4. And we compute the parity C over the entire word (information and SEC parity and DED parity) that was read.

- When an error occurs in a Di bit, what do the check bits c1-c4 contain? What is C? c1-c4 will contain the address of the incorrect bit. The parity C will be 1.
- When an error occurs in a Pi bit, what do the check bits c1-c4 contain? What is C? c1-c4 will contain the address of the incorrect bit. The parity C will be 1.
- When an error occurs in P bit, what do the check bits c1-c4 contain? What is C? c1-c4 will contain 0. C will contain 1.