For this assignment you will develop a Verilog behavioral model that performs a slightly simplified version of floating point multiplication for single precision (32-bit) IEEE format numbers. General information regarding IEEE floating point format can be found at the following link:

http://www.docs.hp.com/en/B3901-90007/ch10s03.html

The single-precision format consists of a sign bit (for the mantissa), an 8-bit exponent stored in excess-127 form, and a 23-bit mantissa. The mantissa is stored as an unsigned magnitude and is always normalized to the form 1.x The leading one to the left of the (implied) radix point is not stored—i.e. the 23-bit stored mantissa represents the fractional part x of the actual 24 bit mantissa.

The excess-127 representation adds a bias of 127 to the stored exponent magnitude. Hence representable exponents range from -127 (represented by the all zero value) to +128 (represented by the all 1’s value). The two extreme exponent values are reserved for special purposes, so valid, non-zero numbers will have exponents in the range from -126 through +127. For our purposes, any number with an exponent > 127 will be considered to be an overflow situation, and any non-zero value with an exponent less than -126 will be considered to be an underflow. Note that the number zero is represented by a stored exponent value of zero (representing an exponent of -127) and a stored mantissa value of zero. IEEE format allows for distinct values +0 and -0, depending on the sign bit.

### IEEE Floating Point Values (b = bias)

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent (e)</th>
<th>Fraction (f)</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00...00</td>
<td>00...00</td>
<td>+0</td>
</tr>
<tr>
<td>0</td>
<td>00...00</td>
<td>00...01</td>
<td>Positive Denormalized Real $0.f \times 2^{(e-b+1)}$ (Treat these as Invalid inputs and as underflow for output)</td>
</tr>
<tr>
<td>0</td>
<td>00...01</td>
<td>. .</td>
<td>Positive Normalized Real $1.f \times 2^{(e-b)}$</td>
</tr>
<tr>
<td>0</td>
<td>11...10</td>
<td>XX...XX</td>
<td>+Infinity (Treat as invalid input)</td>
</tr>
<tr>
<td>0</td>
<td>11...11</td>
<td>00...00</td>
<td>+Infinity (Treat as invalid input)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>
| 0 | 11...11 | 00...01 | sNaN  
(Invalid input) |
| 0 | 11...11 | 10...00 | QNaN  
(Invalid input) |
| 1 | 00...00 | 00...00 | -0 |
| 1 | 00...00 | 00...01 | Negative Denormalized Real  
$-0.f \times 2^{-(b+1)}$  
(Treat these as Invalid inputs and as underflow for output) |
| 1 | 00...01 | XX...XX | Negative Normalized Real  
$-1.f \times 2^{(e-b)}$ |
| 1 | 11...11 | 00...00 | -Infinity  
(Treat as invalid input) |
| 1 | 11...11 | 00...01 | sNAN  
(Invalid input) |
| 1 | 11...11 | 10...00 | QNaN  
(Invalid input) |

Note:  $b$ is the exponent bias. For single-precision, $b = 127$.  

Your floating point multiply module should have two 32-bit inputs, A and B, assumed to supply IEEE format floating point operand values. The module should have a 32-bit output C to convey the result of the floating point operation $A \times B$. The module should also have three one-bit outputs, denoted V, U, and I, respectively. The V output should be set if a floating point multiply results in an overflow—i.e. the result, after renormalization, has an exponent $> 127$. The U output should be set if a floating point multiply results in an underflow—i.e. the result is non-zero and, after renormalization, has an exponent $< -126$. The I (Invalid) bit should be set if either or both of the inputs A and B are not valid IEEE format numbers (as defined above).  

In the case where the V bit is set, the C output should be set to the “positive infinity” value, which is represented by the largest positive exponent  ($+128$, represented by all-ones) and a zero stored mantissa. In the case where the U or I bit is set, the result C should be set to
zero. In all other cases, C should represent the valid result of performing the floating point operation A*B.

The basic algorithm for performing the floating point multiplication is as follows:

1) Compute the sign of the result from the sign bits of the source operands
2) Add the exponents of the source operands to produce the result exponent (be sure to properly adjust the bias of the result exponent.)
3) Compute the mantissa of the result by multiplying the mantissas of the source operands, after the leading ones have been restored. You can treat the source mantissas as 24 bit registers and use the Verilog multiplication operator (*) to multiply them. Note that the result will need to be assigned to a 48-bit register in order to prevent high-order truncation of the result.
4) Check to see if the result mantissa requires a renormalization. If so, shift it right one bit and add one to the result exponent.
5) Assemble the sign, exponent and mantissa to form the result, in proper IEEE format.

Of course, you will need to add checks for the V, U, and I conditions to the above algorithm.

You do not need to implement any form of rounding to account for the loss of significance in the result mantissa (although a “real” IEEE-compliant multiplier would certainly need to do so).

Additional important specifications:

1) Your multiplier module should have the following module definition:
   module fpMult(A,B,C, V,U, I);
   where A,B,C, V,U, and I are as described above. You must conform exactly to this specification since we will be testing your module with our own test fixture.
2) Timing: Your module should function as a combinational multiplier with a propagation delay of 100 time steps—i.e. the outputs (C, V, U, I) should change 100 time steps after any change in either of the inputs A or B.

You should design your own Verilog test bench to thoroughly exercise and test your fpMult module prior to submission. Be sure to test all of combinations of positive, zero, and negative operands with positive, zero, and negative exponents as well as invalid inputs and conditions resulting in overflow or underflow. This test bench is strictly for your own testing purposes and should not be submitted. We will use our own test bench to evaluate your submitted fpMult module.

Submission instructions:

You are to submit only the Verilog source code for your fpMult module. Do NOT submit the code for your test bench or any other artifacts. Your fpMult module should be in a text file
named <hawk-ID>.v  For instance, if your hawk-ID is smith, your file should be called smith.v
You should submit this file as an attachment to an e-mail message sent to:

h pca@engineering.uiowa.edu

To avoid late penalties, your e-mail submission must be time stamped no later than 11:59
p.m. on Monday March 1.