Verilog Fixed point math library

Original work by Sam Skalicky, originally found here:
http://opencores.org/project,fixed_point_arithmetic_parameterized

Extended, updated, and heavily commented by Tom Burke (tomburkeii@gmail.com)

This library includes the basic math functions for the Verilog Language,
for implementation on FPGAs.

All units have been simulated and synthesized for Xilinx Spartan 3E devices
using the Xilinx ISE WebPack tools v14.7

These math routines use a signed magnitude Q,N format, where N is the total
number of bits used, and Q is the number of fractional bits used. For
instance, 15,32 would represent a 32-bit number with 15 fractional bits,
16 integer bits, and 1 sign bit as shown below:

```
|1|<- N-Q-1 bits ->|<--- Q bits -->|
|S|IIIIIIIIIIIIIII|FFFFFFFFFFFFFFF|
```

This library contains the following modules:

```
qadd.v       - Addition module; adds 2 numbers of any sign.
qdiv.v       - Division module; divides two numbers using a right-shift and
                 subtract algorithm - requires an input clock
qmult.v      - Multiplication module; purely combinational for systems that
                 will support it
qmults.v     - Multiplication module; uses a left-shift and add algorithm -
                 requires an input clock (for systems that cannot support
                                 the synthesis of a combinational multiplier)
Test_add.v  - Test fixture for the qadd.v module
Test_mult.v - Test fixture for the qmult.v module
TestDiv.v   - Test fixture for the qdiv.v module
TestMultS.v - Test fixture for the qmults.v module
```

These math routines default to a (Q,N) of (15,32), but are easily customizable
to your application by changing their input parameters. For instance, an
unmodified use of (15,32) would look something like this:

```
     qadd my_adder(
          .a(addend_a),
          .b(addend_b),
          .c(result)
          );
```

To change this to an (8,23) notation, for instance, the same module would be
instantiated thusly:

```
     qadd #(8,23) my_adder(
          .a(addend_a),
          .b(addend_b),
          .c(result)
          );
```

~~~~~~
The following is a description of each module
~~~~~~

qadd.v - A simple combinational addition module.

```
sum = addend + addend

Input format:
|1|<- N-Q-1 bits ->|<--- Q bits -->|
|S|IIIIIIIIIIIIIII|FFFFFFFFFFFFFFF|

Inputs:
     a - addend 1
     b - addend 2

Output format:
|1|<- N-Q-1 bits ->|<--- Q bits -->|
|S|IIIIIIIIIIIIIII|FFFFFFFFFFFFFFF|

Output:
     c - result

NOTE:  There is no error detection for an overflow.  It is up to the designer
         to ensure that an overflow cannot occur!!

Example usage:
     qadd #(Q,N) my_adder(
          .a(addend_a),
          .b(addend_b),
          .c(result)
          );

For subtraction, set the sign bit for the negative number. (subtraction is
the addition of a negative, right?)

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
qmult.v - A simple combinational multiplication module.

Input format:
|1|<- N-Q-1 bits ->|<--- Q bits -->|
|S|IIIIIIIIIIIIIII|FFFFFFFFFFFFFFF|

Inputs:
     i_multiplicand - multiplicand
         i_multiplier   - multiplier

Output format:
|1|<- N-Q-1 bits ->|<--- Q bits -->|
|S|IIIIIIIIIIIIIII|FFFFFFFFFFFFFFF|

Output:
     o_result - result
         ovr      - overflow flag

NOTE:  This module assumes a system that supports the synthesis of
       combinational multipliers.  If your device/synthesizer does not
       support this for your particular application, then use the
       "qmults.v" module.

NOTE:  Notice that the output format is identical to the input format!  To
       properly use this module, you need to either ensure that you maximum
           result never exceeds the format, or incorporate the overflow flag
           into your design

Example usage:
```

```
     qmult #(Q,N) my_multiplier(
          .i_multiplicand(multiplicand),
          .i_multiplier(multiplier),
          .o_result(result),
          .ovr(overflow_flag)
          );

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
qmults.v - A multi-clock multiplication module that uses a left-shift
           and add algorithm.

result = multiplicand x multiplier

Input format:
|1|<- N-Q-1 bits ->|<--- Q bits -->|
|S|IIIIIIIIIIIIIII|FFFFFFFFFFFFFFF|

Inputs:
     i_multiplicand - multiplicand
         i_multiplier   - multiplier
         i_start        - Start flag; set this bit high ("1") to start the
                          operation when the last operation is completed.  This
                          bit is ignored until o_complete is asserted.
         i_clk          - input clock; internal workings occur on the rising
edge

Output format:
|1|<- N-Q-1 bits ->|<--- Q bits -->|
|S|IIIIIIIIIIIIIII|FFFFFFFFFFFFFFF|

Output:
     o_result_out - result
         o_complete   - computation complete flag; asserted ("1") when the
                        operation is completed
         o_overflow   - overflow flag; asserted ("1") to indicate that an
                        overflow has occurred.

NOTE:  This module is "time deterministic ." - that is, it should always
       take the same number of clock cycles to complete an operation,
       regardless of the inputs (N+1 clocks)

NOTE:  Notice that the output format is identical to the input format!  To
       properly use this module, you need to either ensure that you maximum
       result never exceeds the format, or incorporate the overflow flag
       into your design

Example usage:
     qmults #(Q,N) my_multiplier(
          .i_multiplicand(multiplicand),
          .i_multiplier(multiplier),
          .i_start(start),
          .i_clk(clock),
          .o_result(result),
          .o_complete(done),
          .o_overflow(overflow_flag)
          );

The qmults.v module begins computation when the start conditions are met:
     o_complete == 1'b1;
     i_start == 1'b1;
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
qdiv.v - A multi-clock division module that uses a right-shift
           and add algorithm.

quotient = dividend / divisor

Input format:
|1|<- N-Q-1 bits ->|<--- Q bits -->|
|S|IIIIIIIIIIIIIII|FFFFFFFFFFFFFF|

Inputs:
     i_dividend  - dividend
         i_divisor   - divisor
         i_start     - Start flag; set this bit high ("1") to start the
                       operation when the last operation is completed.  This
                       bit is ignored until o_complete is asserted.
         i_clk       - input clock; internal workings occur on the rising edge

Output format:
|1|<- N-Q-1 bits ->|<--- Q bits -->|
|S|IIIIIIIIIIIIIII|FFFFFFFFFFFFFF|

Output:
     o_quotient_out - result
     o_complete     - computation complete flag; asserted ("1") when the
                      operation is completed
     o_overflow     - overflow flag; asserted ("1") to indicate that an
                      overflow has occurred.

NOTE:  This module is "time deterministic ." - that is, it should always
       take the same number of clock cycles to complete an operation,
       regardless of the inputs (N+Q+1 clocks)

NOTE:  Notice that the output format is identical to the input format!  To
       properly use this module, you need to either ensure that you maximum
       result never exceeds the format, or incorporate the overflow flag
       into your design

Example usage:
     qdiv #(Q,N) my_divider(
          .i_dividend(dividend),
          .i_divisor(divisor),
          .i_start(start),
          .i_clk(clock),
          .o_quotient_out(result),
          .o_complete(done),
          .o_overflow(overflow_flag)
          );

The qdiv.v module begins computation when the start conditions are met:
     o_complete == 1'b1;
     i_start == 1'b1;
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Suggestions, Kudos, or Complaints?  Feel free to contact me - but remember,
this stuff is free!
```