LSU EE 4755 Homework 2 Due: 29 September 2023

For instructions visit `https://www.ece.lsu.edu/koppel/v/proc.html`. *For the complete Verilog for this assignment without visiting the lab follow* `https://www.ece.lsu.edu/koppel/v/2023/hw02.v.html`.

Collaboration Rules

Each student is expected to complete his or her own assignment. It is okay to work with other students and to ask questions in order to get ideas on how to solve the problems or how to overcome some obstacle (be it a question of Verilog syntax, interpreting error messages, how a part of the problem might be solved, etc.) It is also acceptable to seek out digital design resources for help on Verilog, digital design, etc. It is okay to make use of AI LLM tools such as ChatGPT and Copilot to generate sample Verilog code. (Do not assume LLM output is correct. Treat LLM output the same way one might treat legal advice given by a lawyer character in a movie: it may sound impressive, but it can range from sage advice to utter nonsense.)

After availing oneself to these resources **each student is expected to be able to complete the assignment alone**. Test questions will be based on homework questions and **the assumed time needed to complete the question will be for a student who had solved the homework assignment on which it was based**.

Problem 0: Following instructions at `https://www.ece.lsu.edu/koppel/v/proc.html`, set up your class account (if necessary), copy the assignment, and run the Verilog simulator on the unmodified homework file, `hw02.v`. Do this early enough so that minor problems (*e.g.*, password doesn't work) are minor problems.

Homework Overview

In this assignment modules will be completed to compute the expression $(1-b/c)/a$. For example, if the inputs to one of these modules are $a = 10$, $b = 20$, and $c = 80$, the output would be $(1-20/80)/10 = 0.075$. The inputs are unsigned integers, but the output is floating point. Module parameters provide the widths of the integer inputs and the significand and exponent size of the floating-point output.

In Problem 1 module `comp_p1` is to be completed so that the calculation is foolishly done in the order given by the expression, $(1-b/c)/a$. The floating point conversion and calculation are to be done using Chipware modules. Solving it requires a straightforward application of Verilog techniques for instantiating modules and wiring them together. It also requires an understanding of when and how to convert numbers from integer to floating-point representations.

In Problem 2 module `comp_p2` is to be completed so that the expression is computed much more efficiently (not foolishly as in Problem 1). The expression $(1-b/c)/a$ is to be transformed so that some of the computation can be done by integer arithmetic and in a way that requires less computation precision.

In a correctly completed assignment the testbench will show that module `comp_p2` has greater accuracy, and the synthesis program will show that module `comp_p2` is both faster and less expensive than `comp_p1`. That is, by transforming $(1-b/c)/a$ all factors of interest improve, there's no cost/performance tradeoff to balance! That's why the method used by `comp_p1` is foolish.

Testbench

To compile your code and run the testbench press `F9` in an Emacs buffer in a properly set up account. The testbench will apply inputs to several instantiation of modules `comp_p1` and `comp_p2`. The instantiations differ on the number of bits used for the integer inputs and the format of the floating-point output. The instantiation parameters are shown at the end of the testbench along with a summary of the errors for that module. The output for an unmodified assignment is:

```
Total comp_p1 exp= 7, sig= 6, w= 4:  9258 errors. Err bits: avg   8.83, max  18
Total comp_p1 exp= 7, sig= 8, w= 4:  9207 errors. Err bits: avg  10.60, max  20
Total comp_p1 exp= 8, sig=10, w= 5:  9533 errors. Err bits: avg  13.60, max  25
Total comp_p1 exp= 8, sig=10, w=10:  9918 errors. Err bits: avg  18.49, max  38
Total comp_p1 exp= 8, sig=12, w=10:  9893 errors. Err bits: avg  20.38, max  39
Total comp_p2 exp= 7, sig= 6, w= 4:  9228 errors. Err bits: avg   9.06, max  18
Total comp_p2 exp= 7, sig= 8, w= 4:  9268 errors. Err bits: avg  10.91, max  20
Total comp_p2 exp= 8, sig=10, w= 5:  9529 errors. Err bits: avg  14.04, max  25
Total comp_p2 exp= 8, sig=10, w=10:  9906 errors. Err bits: avg  19.11, max  39
Total comp_p2 exp= 8, sig=12, w=10:  9903 errors. Err bits: avg  21.15, max  41
Total number of errors: 95643
```

The text `exp= 7` shows the value of parameter `w_exp`, etc. To add or change instantiation parameters search for the place where variable `pset` is assigned and edit the initialization of `pset` (and change `npsets` if needed):

```
localparam int npsets = 5; // This MUST be set to the size of pset.
// { w_exp, w_sig, w_int }
localparam int pset[npsets][3] =
           '{
             { 7,  6,  4 },
             { 7,  8,  4 },
             { 8, 10,  5 },
             { 8, 10, 10 },
             { 8, 12, 10 }};
```

The testbench will report on the correctness and accuracy of the output. The output of a module does not need to exactly match a correct output to be considered correct, it just needs to be close enough. Module `comp_p2` is expected to be more accurate, so an output of `comp_p2` can be considered wrong even though the same output of `comp_p2` is considered correct.

The difference between the expected output and the output provided by your module is measured in *error bits (EB)*. Zero error bits means the output exactly matches. When the exponents of the module and expected output are the same the EB is the size (in bits) of a number that would have to be added to one significand (treating it as an integer) to make it equal to the other. For example, an EB of 1 means that a 1-bit number can be added to one significand to make it equal to the other. An EB of 2 means that a two-bit number can be added. If the exponents differ by more than one then the exponent difference is the EB. See routine `conv::err_bits` for details.

For Problem 2 an output with an EB less than 2 is considered correct. For Problem 1 a per-input tolerance is computed and is used to determine if the output is correct. The testbench keeps track of the average and maximum EB for each module, and these are shown at the end of execution along with an error count. The output for a correct solution is:

```
Total comp_p1 exp= 7, sig= 6, w= 4:     0 errors. Err bits: avg   0.37, max   4
Total comp_p1 exp= 7, sig= 8, w= 4:     0 errors. Err bits: avg   0.40, max   4
Total comp_p1 exp= 8, sig=10, w= 5:     0 errors. Err bits: avg   0.48, max   5
Total comp_p1 exp= 8, sig=10, w=10:     0 errors. Err bits: avg   0.71, max  10
Total comp_p1 exp= 8, sig=12, w=10:     0 errors. Err bits: avg   0.71, max   9
Total comp_p2 exp= 7, sig= 6, w= 4:     0 errors. Err bits: avg   0.00, max   0
Total comp_p2 exp= 7, sig= 8, w= 4:     0 errors. Err bits: avg   0.00, max   0
Total comp_p2 exp= 8, sig=10, w= 5:     0 errors. Err bits: avg   0.00, max   0
Total comp_p2 exp= 8, sig=10, w=10:     0 errors. Err bits: avg   0.07, max   1
Total comp_p2 exp= 8, sig=12, w=10:     0 errors. Err bits: avg   0.04, max   1
```

```
Total number of errors: 0
```

Notice that both modules have zero errors, but that instances of `comp_p2` are more accurate (lower EB). The maximum error bits occurred for `comp_p1` instantiated with a significand width of 10 bits and an integer width of 10 bits. The average EB though is just 0.71, so those big 10-bit errors don't occur very often.

To help in debugging details of errors are shown. Here are the first two errors shown for `comp_p1` with the unmodified code:

```
Error p1 #(7,6,4) a= 1 b=13 c= 1:  Err bits 8 (tol 2)
  Output 2.0000e+00 != -1.2000e+01 (correct).
  Output 'h00 * 2^( 64-63) != 'h20 * 2^( 66-63) (correct)
Error p1 #(7,6,4) a= 5 b=10 c= 5:  Err bits 11 (tol 2)
  Output 6.0000e+00 != -1.9922e-01 (correct).
  Output 'h20 * 2^( 65-63) != 'h26 * 2^( 60-63) (correct)
```

The first list of each error shows the instantiation size (7,6,4), inputs (a=1, b=13,c=1), the EB value, 8, and the tolerance, 2. The tolerance of 2 indicates that an EB of 2 or lower would have been considered correct, but alas the EB is 8. The next two lines (starting with `Output`) show the provided and correct output, in decimal (the first line) and in binary scientific notation (the second line). These lines show for the first error that the expected correct output is -12, but the provided output is 2. The second line shows the significand (in hex) and exponent of the provided and correct output.

Details are not shown for every incorrect output. Instead, details are shown if the EB exceeds the highest EB encountered for that module.

Helpful Examples

For this assignment Chipware modules are to be instantiated to perform floating-point computation and integer/floating-point conversion. See 2017 Homework 2 for examples of how to instantiate these modules to perform a computation and integer/floating-point conversion. In the 2017 assignment all FP numbers were IEEE single 32-bit format. But in this (2023) assignment the formats vary and so parameters must be used when instantiating the Chipware modules to specify the exponent and significand length. In 2021 Homework 2 Chipware modules were instantiated with non-default exponent and significand lengths. Also see 2022 Homework 5. That assignment uses both combinational and sequential modules. (Sequential material has not yet been covered.) See `ms_comb` in 2022 Homework 5 for a straightforward connection of FP modules (but without format conversion).

Problem 1: Module `comp_p1` has three `w_int`-bit integer inputs, `a`, `b`, and `c`, and a `wfp`-bit floating-point output, `h`. The module has three parameters, `w_int`, `w_exp`, and `w_sig`. (A fourth parameter, `wfp` is set to `1+w_exp+w_sig` and its value should not be changed.) Complete module `comp_p1` so that `h` is set to the value of $(1 - b/c)/a$. The module inputs, `a`, `b`, and `c` are unsigned integers but the calculation must be done in floating-point in this problem. Output `h` is a floating-point number with a `w_exp`-bit exponent, a `w_sig`-bit significand, and one sign bit. The format of `h` is the same as the format used by the Chipware modules.

In the unmodified code `comp_p1` computes `h = a + 1`, which is clearly wrong but it does show a quick example of how to convert `a` to floating point, how to get a FP constant, and how to instantiate a Chipware adder.

Complete `comp_p1` so that it foolishly computes `h` based on the calculation order in the expression $\frac{1-b/c}{a}$. (The foolishness is avoided in Problem 2.) That is, first compute $x_1 = b/c$, then compute $x_2 = 1 - x_1$, and finally compute $h = x_2/a$.

Use Chipware modules for the floating-point arithmetic and for conversions between integer and floating-point representations. Pay attention to cost.

A correct solution should show zero errors, but the average bit error can be 0.5 and the maximum bit error can be larger than 5. Lower error rate *and lower cost and lower delay* will be possible in Problem 2.

- ☐ Use Chipware modules for floating-point computation.
- ☐ Use procedural or implicit structural code for any integer computation.
- ☐ Pay attention to cost: The significand size of the floating-point units can be at most `w_sig+1` bits. To achieve this one must provide parameter inputs to the Chipware modules.
- ☐ Pay attention to cost: don't use more bits than are needed.
- ☐ The modules must be synthesizable.

To synthesize your code issue the command `genus -files syn.tcl`. Synthesis should take two or three minutes. If there are no errors, running this command will generate output that includes like the following:

```
Synthesizing at effort level "high"

Module Name                          Area   Delay   Delay     Synth
                                            Actual  Target     Time
comp_p1_w4_w_exp7_w_sig6           183394   31.57   900.0 ns    66 s
comp_p2_w4_w_exp7_w_sig6           129109   18.07   900.0 ns    36 s
```

Problem 2: Expression $\frac{1-b/c}{a}$ might be easy for a human to read, but it does not describe the best way to compute the value with finite-precision computations on non-zero cost hardware. One place accuracy is lost is computing $1 - \frac{b}{c}$ when $b/c \approx 1$. Furthermore all computation must be done in floating-point. Fortunately it is easy to transform $\frac{1-b/c}{a}$ to eliminate the $1 - \frac{b}{c}$ calculation and also to put it in a form where some computation can be done using integer arithmetic. One possible way of transforming the expression is to multiply by 1. Not just any 1 of course, but $\frac{c}{c}$. A few further manipulations should bring it to a form that can be more easily computed.

Module `comp_p2` has the same ports and parameters as `comp_p1`. Complete `comp_p2` so that it computes $\frac{1-b/c}{a}$ much more efficiently, following the guidelines described above. When transforming the expression keep in mind that integer addition and subtraction is less costly than floating-point

subtraction and division (floating-point or integer) is much more costly (time and area) than other operations.

Module `comp_p2` should use a mix of integer and floating-point computation. Pay attention to precision, especially for integer arithmetic where the result of a computation can require more bits than the operands. (If you don't remember try looking it up.)

The testbench applies a stricter test to the output of `comp_p2`, which affects the expected output for inputs in which $b \approx c$.

- ☐ Use Chipware modules for floating-point computation.
- ☐ Use procedural or implicit structural code for integer computation.
- ☐ Pay attention to cost: The significand size of the floating-point units can be at most `w_sig+1` bits. To achieve this one must provide parameter inputs to the Chipware modules.
- ☐ Pay attention to cost: don't use more bits than are needed.
- ☐ The modules must be synthesizable. (Use the same synthesis command as used in Problem 1.)