LSU EE 4755 Homework 2 Solution Due: 4 October 2024

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/2024/hw02.v.html`. *The solution Verilog is in the assignment directory.*
For an htmlized version visit `https://www.ece.lsu.edu/koppel/v/2024/hw02-sol.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 that compute $a2^s + b > c$ where inputs a and b are real and inputs s and c are non-negative integers. Each module has an output `gt`, which should be set to `1` if the comparison is true and `0` otherwise. There is also an output `ssum` which should be set to $a2^s + b$. What makes this interesting is that the sizes of all inputs are parameters, and that in the instantiations tested the number of bits in the significands of a and b can be less than the number of bits in c.

The floating point calculations and conversion(s) are to be done using Chipware modules. Solving this assignment 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 floating-point to integer representations.

As of this writing two modules are to be completed, `comp_fp` and `comp_int`. In `comp_fp` the greater-than comparison is to be done in floating point (using a Chipware module) and in `comp_int` it is to be done using an integer comparison (using the `>` operator).

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_fp` and `comp_int`. 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 end of the testbench output for an unmodified assignment appears below:

```
Total comp_int exp=7, sig=6, wc= 6, s=0: Errors: 50000 ss, 31394 gt.
```

```
Total comp_int exp=7, sig=6, wc= 6, s>0: Errors: 50000 ss, 28962 gt.
Total comp_int exp=7, sig=7, wc=10, s=0: Errors: 50000 ss, 33366 gt.
Total comp_int exp=7, sig=7, wc=10, s>0: Errors: 50000 ss, 31052 gt.
Total comp_int exp=8, sig=5, wc=12, s=0: Errors: 50000 ss, 35958 gt.
Total comp_int exp=8, sig=5, wc=12, s>0: Errors: 50000 ss, 33117 gt.
Total comp_fp  exp=7, sig=6, wc= 6, s=0: Errors: 50000 ss, 31310 gt.
Total comp_fp  exp=7, sig=6, wc= 6, s>0: Errors: 50000 ss, 29113 gt.
Total comp_fp  exp=7, sig=7, wc=10, s=0: Errors: 50000 ss, 33478 gt.
Total comp_fp  exp=7, sig=7, wc=10, s>0: Errors: 50000 ss, 30957 gt.
Total comp_fp  exp=8, sig=5, wc=12, s=0: Errors: 50000 ss, 35987 gt.
Total comp_fp  exp=8, sig=5, wc=12, s>0: Errors: 50000 ss, 33073 gt.
TOOL: xrun(64) 24.03-s005: Exiting on Sep 29, 2024 at 14:00:48 CDT  (total: 00:00:02)

Compilation finished at Sun Sep 29 14:00:48, duration 2.14 s
```

Each line starting with `Total` shows a tally of results. After `Total` the line shows the module name, either `comp_int` or `comp_fp`, and three parameter values. The label `exp` shows the value of parameter `w_exp`, which is the size of the exponent of the FP numbers; label `sig` shows the value of parameter `w_sig`, which the size of the significand of inputs `a` and `b`, and `wc` shows the value of parameter `w_c`, the number of bits in input `c`. The lines with label `s=0` show the results of tests in which module input `s` is set to zero, the lines with label `s>0` show the results of tests in which module input `s` can be non-zero. Tallies of errors are shown after `Errors:`, first of the `ssum` output (scaled sum), and then for the `gt` output. In the unmodified assignment the `ssum` is unconnected, and so its output is always wrong. Output `gt` is set to 1, which is mostly but not always wrong.

Further up, the testbench shows some examples of incorrect output:

```
Starting comp_int tests iwth exp=7, sig=6, wc=6
Error  in #(7,6,6) a=-17.50, s=0, b=0.04, c=3. ss 0.0000e+00 != -1.7464e+01 (correct)
Error  in #(7,6,6) a=-17.50, s=0, b=0.04, c=3. gt 1 != 0 (correct) -20.4644
Error  in #(7,6,6) a=-0.80, s=0, b=18.25, c=12. ss 0.0000e+00 != 1.7445e+01 (correct)
Error  in #(7,6,6) a=12.62, s=0, b=13.62, c=0. ss 0.0000e+00 != 2.6250e+01 (correct)
Error  in #(7,6,6) a=-3.62, s=0, b=3.72, c=0. ss 0.0000e+00 != 9.3750e-02 (correct)
```

In the sample above the first `Error` line indicates that the module output was `0.0000e+00` (that's what a `z` would look like) but $-1.7464 \times 10^1 = -17.464 =$ `-1.7454e+01` was expected. The second `Error` line indicates that the `gt` output was `1` but should have been `0`. The `-20.4544` is the correct difference between $c = 3$ and $-17.5 + 0.04 = -17.46$, indicating that it was not even close. Note that the number of digits past the decimal point is limited and so the full number is not shown. About the first five errors of each type will be shown.

Whether or not there are errors, at least one pair of lines is printed for each test. That output is preceded by the word `Sample` if the output is correct. Appearing below is the beginning and end of the output for correct modules:

```
Starting comp_int tests iwth exp=7, sig=6, wc=6
Sample in #(7,6,6) a=-17.50, s=0, b=0.04, c=3. ss -1.7500e+01 == -1.7464e+01 (cor-
rect)
Sample in #(7,6,6) a=-17.50, s=0, b=0.04, c=3. gt 0 == 0 (correct) -20.4644
Finished  comp_int tests exp=7, sig=6, wc=6, s=0. Errors: 0 ss, 0 gt
Finished  comp_int tests exp=7, sig=6, wc=6, s>0. Errors: 0 ss, 0 gt

[snip]
```

```
Total comp_int exp=7, sig=6, wc= 6, s=0: Errors: 0 ss, 0 gt.
Total comp_int exp=7, sig=6, wc= 6, s>0: Errors: 0 ss, 0 gt.
Total comp_int exp=7, sig=7, wc=10, s=0: Errors: 0 ss, 0 gt.
Total comp_int exp=7, sig=7, wc=10, s>0: Errors: 0 ss, 0 gt.
Total comp_int exp=8, sig=5, wc=12, s=0: Errors: 0 ss, 0 gt.
Total comp_int exp=8, sig=5, wc=12, s>0: Errors: 0 ss, 0 gt.
Total comp_fp  exp=7, sig=6, wc= 6, s=0: Errors: 0 ss, 0 gt.
Total comp_fp  exp=7, sig=6, wc= 6, s>0: Errors: 0 ss, 0 gt.
Total comp_fp  exp=7, sig=7, wc=10, s=0: Errors: 0 ss, 0 gt.
Total comp_fp  exp=7, sig=7, wc=10, s>0: Errors: 0 ss, 0 gt.
Total comp_fp  exp=8, sig=5, wc=12, s=0: Errors: 0 ss, 0 gt.
Total comp_fp  exp=8, sig=5, wc=12, s>0: Errors: 0 ss, 0 gt.
```

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, wc_int }
localparam int pset[npsets][3] =
           '{
             { 7,  6,  4 },
             { 7,  7,  10 },
             { 8,  5,  12 }};
```

The testbench will report on the correctness and accuracy of the output.

References and Helpful Examples

For this assignment Chipware modules are to be instantiated to perform floating-point computation and floating-point/integer conversion. A link to the Genus ChipWare IP Components Guide can be found on the course references page. The IEEE 754 floating point standard is described in the types lecture code, be sure to scroll down to reach it.

See 2023 Homework 2 for examples of how to instantiate these modules to perform a computation and for integer/floating-point conversion. A copy of the solution to the 2023 assignment is included in the 2024 assignment directory. As in this (2024) assignment the floating-point formats in the 2023 assignment 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_fp` has two floating-point (FP) inputs, `a` and `b`, two integer inputs `s` and `c`, one-bit output `gt`, and a FP output `ssum`; it also has integer parameters `w_c`, `w_s`, `w_exp`, `w_sig` and `w_sig2`. (The remaining parameters, `w_fp` and `w_fp2`, are set to the full size of the two FP formats used, don't change them.) Inputs `a` and `b` carry values in a custom IEEE 754 FP format with a `w_exp`-bit exponent and a `w_sig`-bit significand. The total size of each of these inputs is `1+w_exp+w_sig` bits. (The custom format is recognized by the Chipware modules.) The value on input `c` is a `w_c`-bit unsigned integer. The value on output `ssum` is expected to be a IEEE-format FP number with a `w_exp`-bit exponent and `w_sig2`-bit significand.

The output `ssum` (scaled sum) is to be set to $a2^s+b$ and output `gt` is to be set to `1` if $a2^s+b > c$ and `0` otherwise.

For this problem one should review the IEEE 754 notes, plus the use of the Verilog concatenation (like `{2'b11,a,4'd0}`), shift (`a<<2`), and bit slice (`a[6:1]`) operators.

After each subproblem there is a description of how the part was solved. Then the complete solved module is shown.

(a) Modify `comp_fp` so that it computes $a2^s$ *without using Chipware modules*. The value does not have to be assigned to any particular object, but it should be used to compute $a2^s + b$. To solve this subproblem one must understand the IEEE 754 format. A correct solution requires just a line or two of Verilog code. (Just one line if overflow is ignored, which is okay.)

☑ Compute $a2^s$ without Chipware modules.

To compute $a2^s$ for $a \neq 0$ all we need to do is add s to the exponent. For $a = 0$ we would leave the value unchanged. The significand is `w_sig` bits, so to add s to the exponent use expression: `a + ( s << w_sig )` if $a \neq 0$ and `0` otherwise.

Note that for the value $v = f \times 2^e$ with $1 \leq f < 2$ in a representation with a w_e-bit exponent (`w_exp` using the module parameter names) the quantity in the exponent field is $e + e_{\text{bias}}$, where $e_{\text{bias}} = 2^{w_e-1} - 1$. When s is added to the exponent field there is no need to subtract the bias first and then add it again. It is sufficient to just add s. The problem stated that overflow could be ignored.

The line performing the multiplication $a2^s$ appears below:

```
uwire [w_fp-1:0] a_sc = a ? a + ( s << w_sig ) : a;
```

(b) Modify `comp_fp` so that `ssum` is set to $a2^s + b$. (For partial credit, or to get started quickly set `ssum` to $a + b$. If this is done correctly the testbench `s=0` tests should show zero `ss` errors.) Compute the sum using Chipware modules and the value of $a2^s$ from the previous part.

Note that `a` and `b` have a `w_sig`-bit significand, but the sum should have a `w_sig2`-bit significand. So, the significands of $a2^s$ and b must be lengthened (assume that `w_sig2 > w_sig`). See the description of the IEEE 754 format. Please don't look for a module to do this for you.

☑ Convert $a2^s$ and b into FP types with a `w_sig2`-bit significand.

Note that `a` and `b` carry values in a FP representation. All we need to do is widen the significand from `w_sig` bits to `w_sig2` bits. These new bits will be in the least-significant position. The problem did not state what they should be, but a good bet is zero so that the representations of 1.5 and 1.25 don't change. A quick way to do the conversion is to just shift the quantities `w_sig2-w_sig` bits to the left.

```
uwire [w_fp2-1:0] a_cw = a_sc << w_sig2 - w_sig;
uwire [w_fp2-1:0] b_cw = b    << w_sig2 - w_sig;
```

☑ Using the value from the previous part, set `ssum` to $a2^s + b$.

All one has to do is instantiate a ChipWare adder. Examples were shown in the reference assignment 2023 Homework 2. See the code following the parts below.

(*c*) Modify `comp_fp` so that `gt` is correctly set *using a floating-point comparison*. Don't forget that input `c` carries an unsigned integer so that to do a FP comparison `c` will need to be converted.

☑ Set `gt`.

☑ Use Chipware modules for floating-point computation and floating-point/integer conversion.

☑ Use procedural or implicit structural (`assign`) code for any integer computation.

☑ Pay attention to cost: don't use more bits than are needed.

☑ The modules must be synthesizable.

One needs to first convert `c` to a floating point value, and then use a comparison unit to compare it to `ssum`. The comparison unit has many output ports, the only one we need is `agtb`, the others are left unconnected.

```
module comp_fp
  #( int w_c = 5, w_s = 2, w_exp = 5, w_sig = 5, w_sig2 = 6,
     int w_fp = 1 + w_exp + w_sig, w_fp2 = 1 + w_exp + w_sig2 )
   ( output uwire gt,                output uwire [w_fp2-1:0] ssum,
     input uwire [w_fp-1:0] a, b,
     input uwire [w_s-1:0] s,        input uwire [w_c-1:0] c );

   // First, compute a2^s by just adding s to the exponent.
   uwire [w_fp-1:0] a_sc = a ? a + ( s << w_sig ) : a;

   // Convert a_sc and b from FP numbers with w_sig-bit significands to
   // FP numbers with w_sig2-bit significands by just shifting them over.
   uwire [w_fp2-1:0] a_cw = a_sc << w_sig2 - w_sig;
   uwire [w_fp2-1:0] b_cw = b    << w_sig2 - w_sig;

   // Add the now-widened a2^s to b.
   CW_fp_add #( .sig_width(w_sig2), .exp_width(w_exp) )
   d2( .z(ssum), .a(a_cw), .b(b_cw),  .status(), .rnd(Rnd_to_near_up) );

   // Convert c to FP
   uwire [w_fp2:1] cf;
   CW_fp_i2flt #( .sig_width(w_sig2), .exp_width(w_exp), .isize(w_c), .isign(0) )
   coa( .z(cf), .a(c),  .status(), .rnd(Rnd_to_even) );

   // Compare ssum to c
   CW_fp_cmp #( .sig_width(w_sig2), .exp_width(w_exp) )
   cmp( .agtb(gt), .a(ssum), .b(cf),
        .zctr(1'b0), .altb(), .aeqb(), .unordered(),
        .z0(), .z1(), .status0(), .status1() );
   // Note that most of the outputs are unconnected. Hardware
   // for the unconnected outputs will be eliminated.
endmodule
```

Problem 2: Module `comp_int` has the same connections as `comp_fp` and its outputs should be set to the same values.

(*a*) Modify `comp_int` so that it computes `ssum` using an instantiation of `comp_fp`. The `ssum` output of the `comp_fp` instance should connect to the `ssum` output of `comp_int`. Don't use the `gt` output of `comp_fp` so that the synthesis program doesn't synthesize `comp_fp` hardware for `gt`.

☑ Compute `ssum` using an instance of `comp_fp` ☑ with the `gt` output unconnected.

That's just a straightforward instantiation. See the solution code further below.

(*b*) Modify `comp_int` so that `gt` is correctly set *using an integer comparison*. That is, instead of converting `c`, convert `ssum`. For maximum credit convert `ssum` into an integer of as few bits as possible. Don't forget that `c` is unsigned but `ssum` is signed.

☑ Compute `gt` using an integer comparison.

☑ Try to use as few bits as possible.

The value in `c` is `w_c` bits, so to do the comparison as integers one only really needs to convert `ssum` to a `w_c`-bit unsigned integer. The ChipWare `CW_fp_flt2i` however always converts to a signed integer, so we need to perform a `w_c+1`-bit conversion. The maximum representable value is then $2^{w_c} - 1$. What if `ssum` is larger than that, which is easily possible? No problem, just use the status output of `CW_fp_flt2i` to check for an overflow. If there's an overflow then $a2^s + b > c$. As can be determined by reading the ChipWare documentation, there is a overflow when `status[6]=1`. Another special case that needs to be checked is when `ssum` is negative. If it's negative then $a2^s + b > c$ is definitely false. The complete module is shown below.

```
module comp_int
  #( int w_c = 5, w_s = 2, w_exp = 5, w_sig = 5, w_sig2 = 6,
     int w_fp = 1 + w_exp + w_sig, w_fp2 = 1 + w_exp + w_sig2 )
   ( output uwire gt,                output uwire [w_fp2-1:0] ssum,
     input uwire [w_fp-1:0] a, b,
     input uwire [w_s-1:0] s,        input uwire [w_c-1:0] c );

   // Instantiate comp_fp, but leave the gt output unconnected to anything.
   uwire gtx;  // Don't connect this to anything!
   comp_fp #(.w_c(w_c),.w_s(w_s),.w_exp(w_exp),.w_sig(w_sig),.w_sig2(w_sig2))
   fp(gtx,ssum,a,b,s,c);
   // Since gtx is not used the hardware that would connect to gtx isn't synthesized.

   // Convert ssum to a (w_c+1)-bit number.
   uwire [w_c:0] sumi;
   uwire [7:0] sumi_status;
   CW_fp_flt2i #( .isize(w_c+1), .sig_width(w_sig2), .exp_width(w_exp)  ) ftoi
     ( .status(sumi_status), .z(sumi), .a(ssum), .rnd(Rnd_to_plus_inf) );
   // Since c is w_c bits it is wasteful to use more than w_c bits for
   // ssumi, except for the one extra bit used for the sign.

   uwire ssum_positive = !ssum[w_fp2-1],   ssum_overflow = sumi_status[6];

   assign gt = ssum_positive && ( ssum_overflow || sumi > c );
   // Note: If ssum_overflow is true then ssum can't fit in w_c bits and so ssum > c.

endmodule
```

Problem 3: Predict which version will be less expensive, `comp_fp` or `comp_int`. Then run the synthesis program to see which cost less.

Use the command `genus -files syn.tcl` to run synthesis. Be sure to correct any errors that prevent synthesis.

☑ Predict which will be less costly.

Both modules have identical hardware for computing `ssum`, the only difference is in computing `gt`. Remember that in `comp_int` the `gt` output of the `comp_fp` instantiation is left unconnected. That means whatever hardware `comp_fp` would have used to compute `gt` is **not** synthesized when instantiated in `comp_int`.

So, in `comp_fp` the comparison hardware consists of: conversion of a `w_c`-bit integer to a float, and a floating-point comparison of two numbers with `w_sig2`-bit significands.

In `comp_int` a FP value with a `w_sig2`-bit significand is converted into a `w_c`-bit value. The problem says nothing about the relative size of `w_c` and `w_sig2` (and nobody asked), but a look at the testbench reveals they are the same and so one might as well assume that. In `comp_int` the comparison is done as an integer.

The FP comparison in `comp_fp` has to compare both the exponents and significands. That requires examining a total of `w_sig2 + w_exp` bits. The corresponding comparison in `comp_int` need only look at `w_c` bits, plus a sign bit and overflow bit. That favors `comp_int`.

What about the relative costs of conversion? Converting in either direction requires a shifter, which would be $3w\lceil \lg w \rceil \ \mathrm{u_c}$. Here w is the significand or integer size, which is about the same in both cases. Converting to FP requires a count-leading-zeros operation to determine the exponent.

Based on this I'd call it a coin toss. It would be reasonable for a student to assume that it was not necessary to work out a tight estimate of the conversion units' cost in advance. And since it's a pre-synthesis guess there was no expectation of running experiments to see just how much those conversion units cost. So, based on all of this, I would say that `comp_int` should be less costly.

☑ Run synthesis to find out which really is.

Appearing below are the synthesis results:

```
Module Name                              Area   Delay   Delay     Synth
                                              Actual  Target     Time
comp_int_w_c6_w_s6_w_exp7_w_sig6_w_sig27
                                         148989   23.33   900.0 ns     17 s
comp_fp_w_c6_w_s6_w_exp7_w_sig6_w_sig27
                                         140689   20.42   900.0 ns     13 s
comp_int_w_c12_w_s6_w_exp7_w_sig6_w_sig213
                                         244127   30.05   900.0 ns     19 s
comp_fp_w_c12_w_s6_w_exp7_w_sig6_w_sig213
                                         210213   26.19   900.0 ns     17 s
```

Remember that the cost is shown under `Area` and the critical path length is under `Delay Actual`. The `Synth Time` column shows the amount of time it took to run synthesis for that module, which has no bearing on the synthesized hardware itself.

Looking at the data above I see that ooops, I was wrong. At both sizes the integer units were less costly. This is also true with the synthesis effort levels set to high. (They were medium in the synthesis script.)

The FP delay is also lower. I'm tempted to run further experiments to determine why. But perhaps I'll leave that for a future assignment.