

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/2026/hw02.v.html>.

Student Expectations

To solve this assignment students are expected to avail themselves of references provided in class and on the Web site, such as for Verilog programming and synthesis examples, and to seek out any additional help and resources that might be needed. (Of course this doesn't mean asking someone else to solve it for you.) It is each student's duty to himself or herself to resolve frustrations and roadblocks quickly. (If you get stuck *just ask for help!*)

This assignment cannot be solved by blindly pasting together parts of past assignments. Solving the assignment is a multi-step learning process that takes effort, but one that also provides the satisfaction of progress and of developing skills and understanding.

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, the IEEE 754 floating-point format, etc. It is okay to make use of AI LLM tools such as Claude 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—not memorized the solution of—the homework assignment on which it was based**.

Students are expected to write code that follows the conventions presented in class and that avoids features that are considered deprecated as far as this course is concerned. Code not following these conventions will be assumed to have been copied or LLM-generated.

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

Converting a number from an integer format to a floating-point format sounds straightforward, and it is. Chipware module `CW_fp_i2flt` performs such conversions two FP, and can do so for unsigned and to's complement signed integers. (*Sorry about the puns.*) When this assignment is completed module `tiger_itof` will also convert integers to FP, but it will be able to handle a third format, sign-magnitude, in which the most-significant bit is the sign (1 for negative) and the remaining bits are the absolute value of the number. Module `tiger_itof` is the subject of Problem 1. To solve

Problem 1 one needs a good understanding of how the bias and significand in IEEE 754 are computed. See the material on FP in <https://www.ece.lsu.edu/koppel/v/2026/1020-types.v.html> for details.

Module `tiger_itof` instantiates module `left_justify_wrapper`, which combines a count-leading zeros module and a shift-left module, both reporting the number of leading zeros in its integer input, and shifting the integer so that leading zeros are shifted off. In Problem 2 module `left_justify_custom` is to be completed so that it performs this function at a lower cost than a combined `clz` and shift module could. Problem 2 exercises skills in digital design and the use of generate statements.

Testbench

To compile your code and run the testbench press `[F9]` in an Emacs buffer in a **properly set up account**. The testbench, as of this writing, will test 24 (twenty-four!) instantiations of `tiger_itof`. For each instantiation it will show sample values at the inputs and outputs of `tiger_itof` and `left_justify_wrapper`. It will also show whether those output values are correct, and what those values should be. (See below.) After this detailed information the testbench prints tallies of the number of errors.

Module `tiger_itof` can use two (actually three) different versions of `left_justify_wrapper`, determined by parameter `lj_way`. When `lj_way=M_lj_parts` the module uses `left_justify_parts`, a complete left-justify module. When `lj_way=M_lj_custom` it uses `left_justify_custom`, a module to be completed in this assignment.

Module `tiger_itof` can convert integers in three different formats, determined by parameter `i_fmt`. When `i_fmt=M_i_twos_comp` input `i` is expected to be in 2's complement form, when `i_fmt=M_i_unsigned` input `i` is expected to be unsigned, and when `i_fmt=M_i_sign_mag` input `i` is expected to be in a sign-magnitude format (explained in the Problem 1 description).

The testbench and synthesis script use **the following abbreviations** for these left-justify variations: for `M_lj_parts`: `jlp`, `lj_parts`, `lj_way0`; for `M_lj_custom`: `jlc`, `lj_cust`, `lj_way1`. The testbench and synthesis script use the following abbreviations for integer formats: for `M_i_unsigned`: `unsigned`, `uns`, `ifmt0`; for `M_i_twos_comp`: `twos-comp`, `2sc`, `ifmt1`; for `M_i_sign_mag`: `sign-mag`, `smg`, `ifmt2`.

The modules are also instantiated with different integer widths (`w_i`) and with different floating-point formats (`w_exp`, `w_sig`). One thing to keep in mind is that the integer width may be less than, equal to, or larger than the significand width. Appearing below is are excerpts from the testbench output for a correctly solved solution.

Compilation started at Tue Sep 29 17:52:37

```
xrun -sv -batch -exit hw02.v
```

```
T00L: xrun(64) 26.03-s005: Started on Sep 29, 2026 at 17:52:38 CDT
```

```
[snip]
```

```
xcelium> run
```

```
Starting Int unsigned w_i=3, Left-Just lj_parts, FP Fmt w_exp=7, w_sig=6
Sample ljp uns #(7,6,3) i=7='h7, nlz,lj: {0,'h7} == {0,'h7} (correct)
Sample ljp uns #(7,6,3) s,exp,sig: {0,65,'h30} == {0,65,'h30} (correct)
Sample ljp uns #(7,6,3) i=0='h0, nlz,lj: {3,'h0} == {0,'h0} (correct)
```

```
Sample ljp uns #(7,6,3) s,exp,sig: {0,0,'h00} == {0,0,'h00} (correct)
Finished lj_parts unsigned exp=7, sig=6, wi=3. Errors: 0 nlz, 0 lj, 0 f
```

[snip]

```
Starting Int twos-comp w_i=8, Left-Just lj_parts, FP Fmt w_exp=7, w_sig=8
Sample ljp 2sc #(7,8,8) i=-1='hff, nlz,lj: {7,'h80} == { 7,'h80} (correct)
Sample ljp 2sc #(7,8,8) s,exp,sig: {1,63,'h00} == {1,63,'h00} (correct)
Sample ljp 2sc #(7,8,8) i=0='h00, nlz,lj: {0,'h00} == { 0,'h00} (correct)
Sample ljp 2sc #(7,8,8) s,exp,sig: {0,0,'h00} == {0,0,'h00} (correct)
Finished lj_parts twos-comp exp=7, sig=8, wi=8. Errors: 0 nlz, 0 lj, 0 f
```

[snip]

As seen above, the detailed test for each instantiation starts with a line starting **Starting**. A line starting **Sample** shows sample module outputs when the outputs are correct. At most two pairs of samples are shown. (You can set `n_samples` in the testbench to your preferred number of samples.) The first of a pair of samples has the text `i=` showing the integer input value (such as 7 and 0 above) and has the bit vector in hexadecimal. Note that when an input is negative the value is shown as negative, such as -1, while the bit vector depends on the format. For -1 the bit vector is 'hff.

The line with `i=` shows the outputs of the chosen `left_justify` module, `nlz` and `justified` (labeled as `lj`). The values are shown in braces to the left of the `==` (or `!=`). The values to the right are the correct values of those quantities. In the samples above the module outputs are correct. The `nlz` value is shown in decimal and the `justified` value in hexadecimal.

The line with `s,exp,sig` shows the `f` output of the `tiger_itof` split into sign, exponent, and significand and shown to the left of the `==` (or `!=` if they were wrong). The correct values are on the right.

Appearing below are testbench outputs from an unsolved assignment.

```
Starting Int unsigned w_i=3, Left-Just lj_parts, FP Fmt w_exp=7, w_sig=6
Sample ljp uns #(7,6,3) i=7='h7, nlz,lj: {0,'h7} == {0,'h7} (correct)
Error ljp uns #(7,6,3) s,exp,sig: {0,16,'h07} != {0,65,'h30} (correct)
Sample ljp uns #(7,6,3) i=0='h0, nlz,lj: {3,'h0} == {0,'h0} (correct)
Error ljp uns #(7,6,3) s,exp,sig: {0,0,'h18} != {0,0,'h00} (correct)
Sample ljp uns #(7,6,3) i=6='h6, nlz,lj: {0,'h6} == {0,'h6} (correct)
Error ljp uns #(7,6,3) s,exp,sig: {0,16,'h06} != {0,65,'h20} (correct)
Sample ljp uns #(7,6,3) i=1='h1, nlz,lj: {2,'h4} == {2,'h4} (correct)
Error ljp uns #(7,6,3) s,exp,sig: {0,0,'h14} != {0,63,'h00} (correct)
Finished lj_parts unsigned exp=7, sig=6, wi=3. Errors: 0 nlz, 0 lj, 1000 f
```

[snip]

```
Starting Int unsigned w_i=3, Left-Just lj_cust, FP Fmt w_exp=7, w_sig=6
Error ljc uns #(7,6,3) i=7='h7, nlz,lj: {z,'hz} != {0,'h7} (correct)
Error ljc uns #(7,6,3) s,exp,sig: {0,16,'hZz} != {0,65,'h30} (correct)
Error ljc uns #(7,6,3) i=0='h0, nlz,lj: {z,'hz} != {0,'h0} (correct)
Error ljc uns #(7,6,3) s,exp,sig: {0,0,'hZz} != {0,0,'h00} (correct)
```

```

Error  ljc uns #(7,6,3) i=6='h6, nlz,lj: {z,'hz} != {0,'h6} (correct)
Error  ljc uns #(7,6,3) s,exp,sig: {0,16,'hZz} != {0,65,'h20} (correct)
Error  ljc uns #(7,6,3) i=1='h1, nlz,lj: {z,'hz} != {2,'h4} (correct)
Error  ljc uns #(7,6,3) s,exp,sig: {0,0,'hZz} != {0,63,'h00} (correct)
Error  ljc uns #(7,6,3) i=5='h5, nlz,lj: {z,'hz} != {0,'h5} (correct)
Error  ljc uns #(7,6,3) s,exp,sig: {0,16,'hZz} != {0,65,'h10} (correct)
Finished  lj_cust unsigned exp=7, sig=6, wi=3. Errors: 875 nlz, 1000 lj, 1000 f

```

The first group above uses a correct version of `left_justify`, and an unsigned format so `nlz` and `lj` are correct. The sign, exponent, and significand are wrong. In the second group the `nlz` and `lj` values are `z` because the outputs of `left_justify_custom` were unconnected.

At the end of the testbench output are tallies of the total number of errors. The sample below is from an unsolved assignment.

```

[snip]
Total lj_parts unsigned  Errors: 0 nlz, 0 lj, 2996 f.
Total lj_parts twos-comp Errors: 1371 nlz, 2 lj, 2996 f.
Total lj_parts sign-mag  Errors: 1371 nlz, 2 lj, 2996 f.
Total lj_cust  unsigned  Errors: 2561 nlz, 2 lj, 3000 f.
Total lj_cust  twos-comp Errors: 2252 nlz, 2 lj, 3000 f.
Total lj_cust  sign-mag  Errors: 2150 nlz, 2 lj, 3000 f.
Total All Configs:  Errors: 9705 nlz, 9855 lj, 17988 f.

```

To help you debug it is okay to modify the testbench so that it applies tests that help you discover flaws. The best thing to do is to add additional tests, or move a test so that it is first. If you eliminate tests, be sure to put them back. Note that the TA-bot will use a fresh testbench, not the version in your file.

Synthesis Script

Modules for this assignment should be synthesized using the command `genus -f syn.tcl`. The script synthesizes modules `tiger_itof` (Problem 1) and `left_justify_custom` (Problem 2), as well as `cw_itof`, a module that instantiates the Chipware integer-to-float module, and two other left justify modules. The left-justify modules are synthesized at three different sizes, and the `itof` modules at two different sizes. All modules are synthesized with an easy 100 ns delay target, so the synthesis program can focus on reducing cost. Some modules are also synthesized with a 0.1 ns delay target, prioritizing speed.

Column **Area** shows area, which for us is cost. Column **Delay Actual** is delay of the synthesized module, which is the longest path from input to output. Column **Delay Target** is the maximum acceptable delay we chose. The synthesis program will reduce delay until this delay target is reached, then it switches to optimizing to reduce cost. Column **Synth Time** is how long the synthesis program itself took to run. That matters for impatient engineers and whoever is paying for compute time, but it does not in any way describe the synthesized hardware.

Since we are designing an integer-to-floating-point module it would be helpful to compare our design to a commercial product. That comparison design is module `CW_fp_i2flt` in the Cadence Chipware library. To make it easy to compare designs the Chipware module is instantiated in wrapper module `cw_itof`. Module `cw_itof` has the same parameters and ports as `tiger_itof` so it is easy to test in our testbench and synthesize using our synthesis script. This gives us assurance that we are comparing modules that convert the same integer format to the same FP

format (except for sign magnitude, which the Chipware module does not do). *(Possible exam short answer question: modify `cw_itof` so that it **can** convert sign magnitude.)* Our testbench checks the `left_justify_wrapper` module instantiated by `tiger_itof`. So that `cw_itof` can be checked by the same testbench code, it instantiates `left_justify_wrapper` but does not use its outputs. The instantiation is guarded by `ifdef` directives that include the instantiation under simulation but omit it during synthesis, since the Chipware module does not use `left_justify_wrapper`. This way we can be sure that the cost of `cw_itof` does not include the cost of `left_justify_wrapper`. *Note: In a prior version of the assignment code the `ifdef` guard was omitted. Even so the synthesis program eliminated the `left_justify_wrapper` module because its outputs were not used. So the `cw_itof` costs in the earlier homework handout are the same as this one.*

The output of the synthesis script on a correct solution appears below. Notice that the cost of “our” `tiger_itof` (the `way1` versions) is lower than `cw_itof` which uses the integer-to-float module in the Chipware library. The likely reason for our lower cost is that our modules were written to round towards zero, while the Chipware module has the desired rounding mode as an input port. That input port is connected to an elaboration constant value of `Rnd_to_0`, but the optimizer seemingly did not take advantage of the constant input.

Module Name	Area	Delay Actual	Delay Target	Synth Time
<code>left_justify_custom_w4</code>	2762	0.40	100.0 ns	5 s
<code>tiger_itof_lj_way0_ifmt0_wi7_wexp7_wsig7</code>	10049	1.23	100.0 ns	2 s
<code>tiger_itof_lj_way1_ifmt0_wi7_wexp7_wsig7</code>	8900	1.60	100.0 ns	2 s
<code>cw_itof_lj_way1_ifmt0_wi7_wexp7_wsig7</code>	9836	1.66	100.0 ns	3 s
<code>tiger_itof_lj_way0_ifmt1_wi7_wexp7_wsig7</code>	12159	2.81	100.0 ns	2 s
<code>tiger_itof_lj_way1_ifmt1_wi7_wexp7_wsig7</code>	11841	2.26	100.0 ns	2 s
<code>cw_itof_lj_way1_ifmt1_wi7_wexp7_wsig7</code>	15903	2.84	100.0 ns	4 s
<code>tiger_itof_lj_way0_ifmt2_wi7_wexp7_wsig7</code>	7601	0.82	100.0 ns	2 s
<code>tiger_itof_lj_way1_ifmt2_wi7_wexp7_wsig7</code>	7555	1.35	100.0 ns	2 s
<code>tiger_itof_lj_way0_ifmt0_wi16_wexp7_wsig16</code>	30087	3.91	100.0 ns	4 s
<code>tiger_itof_lj_way1_ifmt0_wi16_wexp7_wsig16</code>	27231	3.80	100.0 ns	3 s
<code>cw_itof_lj_way1_ifmt0_wi16_wexp7_wsig16</code>	31065	2.74	100.0 ns	5 s
<code>tiger_itof_lj_way0_ifmt1_wi16_wexp7_wsig16</code>	44558	5.67	100.0 ns	5 s
<code>tiger_itof_lj_way1_ifmt1_wi16_wexp7_wsig16</code>	42719	6.84	100.0 ns	6 s
<code>cw_itof_lj_way1_ifmt1_wi16_wexp7_wsig16</code>	50673	6.75	100.0 ns	8 s
<code>tiger_itof_lj_way0_ifmt2_wi16_wexp7_wsig16</code>	29111	3.50	100.0 ns	4 s
<code>tiger_itof_lj_way1_ifmt2_wi16_wexp7_wsig16</code>	25768	3.46	100.0 ns	3 s
<code>left_justify_procedural_w4</code>	3433	0.34	100.0 ns	1 s
<code>left_justify_parts_w4</code>	3814	0.38	100.0 ns	1 s
<code>left_justify_custom_w4</code>	2762	0.40	100.0 ns	1 s
<code>left_justify_procedural_w8</code>	13766	1.16	100.0 ns	3 s
<code>left_justify_parts_w8</code>	11414	1.94	100.0 ns	2 s
<code>left_justify_custom_w8</code>	9149	1.50	100.0 ns	2 s
<code>left_justify_procedural_w16</code>	48977	1.79	100.0 ns	6 s
<code>left_justify_parts_w16</code>	29747	3.75	100.0 ns	4 s
<code>left_justify_custom_w16</code>	24984	4.28	100.0 ns	3 s

left_justify_procedural_w4	4491	0.20	0.1 ns	3 s
left_justify_parts_w4	4491	0.20	0.1 ns	3 s
left_justify_custom_w4	3389	0.17	0.1 ns	2 s
left_justify_procedural_w8	23888	0.32	0.1 ns	12 s
left_justify_parts_w8	24873	0.49	0.1 ns	16 s
left_justify_custom_w8	23652	0.48	0.1 ns	21 s
left_justify_procedural_w16	94887	0.58	0.1 ns	31 s
left_justify_parts_w16	53229	1.47	0.1 ns	36 s
left_justify_custom_w16	40072	1.36	0.1 ns	27 s
tiger_itof_lj_way0_ifmt0_wi7_wexp7_wsig7	20082	0.35	0.1 ns	9 s
tiger_itof_lj_way1_ifmt0_wi7_wexp7_wsig7	14377	0.58	0.1 ns	15 s
cw_itof_lj_way1_ifmt0_wi7_wexp7_wsig7	33094	0.46	0.1 ns	20 s
tiger_itof_lj_way0_ifmt1_wi7_wexp7_wsig7	21196	0.89	0.1 ns	16 s
tiger_itof_lj_way1_ifmt1_wi7_wexp7_wsig7	22006	1.07	0.1 ns	29 s
cw_itof_lj_way1_ifmt1_wi7_wexp7_wsig7	45978	1.18	0.1 ns	44 s
tiger_itof_lj_way0_ifmt2_wi7_wexp7_wsig7	14078	0.26	0.1 ns	7 s
tiger_itof_lj_way1_ifmt2_wi7_wexp7_wsig7	13982	0.30	0.1 ns	7 s

References and Helpful Examples

Be sure to understand the material on IEEE 754 covered in

<https://www.ece.lsu.edu/koppel/v/2026/1020-types.v.html>. Also make sure to understand the logarithmic shifter covered earlier in this semester.

Problem 1: Module `tiger_itof` has one input, `i`, and one output `f`. The module, when complete, will set `f` to the IEEE 754-like floating-point representation of integer input `i` based on the round-towards-zero rounding mode. The module parameters indicate the size and format of `i` and `f` and what left-justify module it should use. Parameter `w_i` is the number of bits in `i`. Parameters `w_exp` and `w_sig` are the number of bits in the exponent and significand in the FP representation to be used for `f`.

Parameter `i_fmt` indicates the format of `i`, either unsigned, two-complement, or sign magnitude. Students are expected to know unsigned and two's complement representations. In a sign magnitude representation the most significant bit is the sign, 1 for negative. The remaining bits are the magnitude of the value. For example, consider `w_i=5` (five bits). A value of 3 would be encoded as 00011, a value of -3 would be 10011 in sign magnitude. (In two's complement, 3 is 00011 and -3 is 11101.)

Parameter `lj_way` indicates which left-justify module to use. It is used in the instantiation of `left_justify_wrapper`. Other than that, the value of `lj_way` should not be used in your solution. One can safely ignore it.

Module `tiger_itof` includes an instantiation of `left_justify_wrapper`. This module has an input, `i`, and two outputs, `i_left` and `nlz`. If `i` is non-zero output `nlz` is the number of leading zeros in `i`, otherwise `nlz` can be anything. Output `i_left` is the value of `i` left-shifted so that the first 1 in `i` is the most-significant bit of `i_left` if `i` is non-zero otherwise `i_left` is set to zero. For example, if `i=00101` then `nlz=2` and `i_left=10100`, and if `i=00000` then `i_left=00000` and `nlz` should be ignored. Note that if `i` is zero then the most-significant bit of `i_left` will be 0.

Complete `tiger_itof` so that it assigns `f` the correct value and does so using the output of `left_justify_wrapper`. The input to `left_justify_wrapper` can *and should* be changed. If this problem is solved before Problem 2, then ignore testbench outputs for `lj_way=M_lj_custom`.

For the round-towards-zero rounding mode bits beyond the significand width are truncated. If you were not thinking about rounding modes when you designed the module it would turn out that round-towards-zero is what you'd get. *Yes, I speak from experience. I didn't realize this until I ran `cw_itof` through the testbench with the rounding mode set to the more common round-to-even and found that its answer differed from my solution to `tiger_itof`.*

The goal of this problem is to learn about floating-point formats. It would be super-easy to find Verilog descriptions of a conversion module on the Web or to ask an LLM to generate one for you. That would be a waste of your tuition money and would not help you on the exam.

Follow other guidelines and restrictions in the checklist in the file in the Problem 1 area.

There's another problem on the next page!

Problem 2: Module `left_justify_parts` left-justifies its input by using a `clz` module to count the number of leading zeros in its input and a `shift_left_logarithmic` to shift the first 1 to the most-significant bit position. That works, but is it the best way? Perhaps if the functionality of the `clz` and `shift_left` modules were combined one could get a less expensive or faster result. As of this writing, yes. Appearing below are the cost and delay of three justify module variations synthesized with two delay targets:

Module Name	Area	Delay	Delay	Synth Time
		Actual	Target	
<code>left_justify_procedural_w16</code>	48977	1.79	100.0 ns	6 s
<code>left_justify_parts_w16</code>	29747	3.75	100.0 ns	4 s
<code>left_justify_custom_w16</code>	24984	4.28	100.0 ns	3 s
<code>left_justify_procedural_w16</code>	94887	0.58	0.1 ns	31 s
<code>left_justify_parts_w16</code>	53229	1.47	0.1 ns	36 s
<code>left_justify_custom_w16</code>	40072	1.36	0.1 ns	27 s

Module `left_justify_procedural`, in the homework file, is a straightforward procedural module for left justification. Module `left_justify_custom` shown in the output above is a solution to this problem. As one can tell from the first three values under **Area** it is less expensive than either other module. It is also slowest, but the synthesis program was only told to keep the delay at or below 100 ns. (The values under **Synth Time** show how long it took the synthesis itself program to run.) When the delay target is much tighter, 0.1 ns, the custom module is still cheapest but the procedural code is fastest.

Complete `left_justify_custom` so that it left-justifies at lower cost than the other modules. A good place to start is by copying in code from `shift_left_logarithmic`. Don't copy in `clz` code, instead figure out how much to shift at each stage in the shifter.

The module must not use procedural code and should use generate statements. The testbench will check whether it is correct. To test synthesizability use command `genus -files syn.tcl`.

Follow the guidelines and suggestions in the Problem 2 area of the assignment file.