This document contains assignments given in LSU EE 4755 over many semesters. It was automatically generated and so some solutions (and possibly some assignments) are likely missing. At the top of each page of each assignment is a link to the original assignment. Those who want to print an assignment might follow that link. All assignments and public solutions are available at `https://www.ece.lsu.edu/ee4755/prev.html`.

Contents

1 Fall 2024

LSU EE 4755 Homework 1 Due: 20 September 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/hw01.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, copy the assignment, and run the Verilog simulator on the unmodified homework file, `hw01.v`. Do this early enough so that minor problems (*e.g.*, password doesn't work) are minor problems.

Homework Introduction

In this assignment various modules computing dot products will be completed. The actual computation needed is shown in the comments, so forgetting what a dot product is should not be an impediment to this assignment, but it is something you should know. The dot product of n-element vector a with n-element vector b is given by $a \cdot b = \sum_{i=0}^{n-1} a_i b_i$.

The modules for Problem 1, `dot2`, `dot3`, `dot4`, compute the dot product efficiently. That's why they are first. The modules for Problem 2, `dot2m`, `dot4m`, and `dot6m`, compute the same result, but the way in which the data flows through the modules is different (and may result in slower hardware if the synthesis program does not come to the rescue). The modules in Problem 3 share the tree structure of those in Problem 1, but precision of the two input vectors can be different, for example one vector can consist of 8-bit elements and the other of 4-bit elements. The pedagogical motivation is to exercise skill with module parameters, but it also is good exposure to an important feature of hardware used for neural network computations: mixed precision.

Testbench

To compile your code and run the testbench press F9 in an Emacs buffer in a properly set up account. (Of course the testbench won't run until compilation errors are fixed.) The testbench will test modules for all three problems in this assignment. The beginning of the testbench output, which may quickly scroll by, will look something like this:

```
Compilation started at Sat Sep 14 17:18:51

xrun -sv -batch -exit hw01.v
TOOL: xrun(64) 24.03-s005: Started on Sep 14, 2024 at 17:18:51 CDT
xrun(64): 24.03-s005: (c) Copyright 1995-2024 Cadence Design Systems, Inc.
Recompiling... reason: file './hw01.v' is newer than expected.
```

```
expected: Sat Sep 14 17:06:07 2024
actual:   Sat Sep 14 17:18:50 2024
```

At the end of the testbench output is a tally of the number of errors in each module. For a correctly solved assignment the output will be:

```
End of tests. For dot2:  0 errors out of 2000 tests.
End of tests. For dot2m:  0 errors out of 2000 tests.
End of tests. For dot2y:  0 errors out of 2000 tests.
End of tests. For dot3:  0 errors out of 2000 tests.
End of tests. For dot4:  0 errors out of 2000 tests.
End of tests. For dot4m:  0 errors out of 2000 tests.
End of tests. For dot4y:  0 errors out of 2000 tests.
End of tests. For dot6m:  0 errors out of 2000 tests.
xmsim: *W,RNQUIE: Simulation is complete.
xcelium> exit
TOOL: xrun(64) 24.03-s005: Exiting on Sep 14, 2024 at 17:18:52 CDT  (total: 00:00:01)
```

Further up in the output the testbench shows the details for modules that produced incorrect output. The testbench will show the name of the module, the parameters (`w` or `wa` and `wb`), the module output, the expected value (just to the right of "(correct)". On the following lines the input to the module will be shown, first in decimal, then in hexadecimal. For example, in an unmodified assignment the first reported errors are:

```
Starting tests for Prob 1 w=2.
Error, dot2, w=2, z != 0 (correct)
 0 * 0 + 0 * 0
 0x0 * 0x0 + 0x0 * 0x0
Error, dot2, w=2, z != 0 (correct)
 0 * 1 + 0 * 1
 0x0 * 0x1 + 0x0 * 0x1
```

The output of the module here is `z`, meaning it isn't connected to anything.

The testbench starts off with debug-friendly inputs. In the first input all elements of `a` and `b` are zero (seen above), in the second all elements of `a` are zero and elements of `b` are 1. After about six debug-friendly sets of inputs the inputs are randomly chosen. If you'd like to set your own debug-friendly inputs, search for "Test Patterns" and add (or modify) one of the cases. It would be a good idea not to modify the default case. For grading, your assignment will be tested with a fresh copy of the testbench.

Here are some more examples of errors reported by the testbench:

```
Starting tests for Prob 1 w=2.
Error, dot2, w=2, 0 != 1 (correct)
 1 * 0 + 1 * 1
 0x1 * 0x0 + 0x1 * 0x1
Error, dot2, w=2, 1 != 3 (correct)
 1 * 3 + 2 * 0
 0x1 * 0x3 + 0x2 * 0x0
Error, dot2, w=2, 1 != 3 (correct)
 2 * 1 + 3 * 3
 0x2 * 0x1 + 0x3 * 0x3
```

The testbench will only show details of the first five errors in each module.

Common Problems

Here are some common errors messages, which will be encountered when the code is compiled (for example, by pressing F9).

```
file: hw01.v
   mult mym( p, a[0], b[0] );
              !
xmvlog: *E,NODFNT (hw01.v,105!13): Implicit net declaration (p) is NOT allowed, since
'default_nettype is declared as NONE [19.2(IEEE 2001)].
```

The problem above is that `p` was never declared. "Implicit net declaration" refers to a Verilog feature in which an object is assumed to be a `wire` if it had not been declared. That feature has been turned off since it can hide typos. The solution is to declare something like `uwire [um-1:0] p;`.

```
   mult mym( p, a[0], b[0] );
                 !
mislabel: *F,CUVMPW (./hw01.v,106!16): port sizes differ in port connection(8/5) for
the instance(testbench.genblk1[1].tb.genblk1.d2) .
```

The problem above is that the number of bits in `a[0]` is not the same as the number of bits in the second input to `mult`. The solution is to specify the mult `w` parameter value in the instantiation.

Helpful Examples

A good past assignment to look at is 2023 Homework 1. Like this assignment, a module, say `minmax8`, is completed by instantiating other modules and splitting `minmax8`'s inputs among the instantiations. The tree-like structure of the modules in Problem 1 and 3 of this (2024) assignment matches those in the 2023 assignment. Problem 2 is sort of tree-like. Problem 3 in this assignment requires more attention to parameters and port sizes than in past assignments.

Problem 1: Modules `dot2`, `dot3`, and `dot4`, each have two inputs, `a` and `b`, and one output `dp`, and parameter `w`. In all modules output `dp` is a `w`-bit value. In `dot2` inputs `a` and `b` are each 2-element arrays of `w`-bit values. In `dot3` and `dot4` inputs `a` and `b` are 3- and 4-element arrays of `w`-bit values. The inputs and outputs are (to be interpreted as) unsigned integers. Output `dp` is to be set to the dot product of vectors `a` and `b` (the computation is shown in the comments of each module).

(*a*) Complete module `dot2` using instantiations of modules `mult` and `add`. These modules are shown below and are in the Problem 1 part of the assignment file.

```
module mult
  #( int w = 5 ) ( output uwire [w-1:0] p, input uwire [w-1:0] a, b );
   assign p = a * b;
endmodule

module add
  #( int w = 5 ) ( output uwire [w-1:0] s, input uwire [w-1:0] a, b );
   assign s = a + b;
endmodule

module dot2
  #( int w = 5 )
   ( output uwire [w-1:0] dp,
     input uwire [w-1:0] a[1:0], b[1:0] );

   // Compute
   //   dp = a[0] * b[0] + a[1] * b[1];
   // Using instantiation(s) of
   //  mult, add.

endmodule
```

Do not use procedural code and do not use continuous assignments in `dot2` and in the other modules to be completed in this assignment. Continuous assignments start with the `assign` keyword, and procedural code starts with `always_comb`, `always`, `initial`, etc. In addition to instantiating `add` and `mult` modules it will be necessary to declare `uwire` objects. These requirements are listed in the checkbox items in the code.

(*b*) Complete module `dot3`. As with `dot2`, procedural code and continuous assignments cannot be used. However, to complete `dot3` use an instantiation of `dot2` and as many `add` and `mult` modules as needed. (Obviously the fewer the better.) These requirements are listed in the checkbox items in the code.

(*c*) Complete module `dot4`. Consider instantiations of modules `add`, `mult`, `dot2`, and `dot3`. There are several ways to solve this. Instantiate as few modules as is reasonable and also consider how long it will take to compute the result. (The time to compute a result has not been covered, but one should be able to get a feel for it by drawing a diagram of the hardware.)

If the instructions above were followed the modules should be synthesizable. This can be verified by using the command `genus -files syn.tcl`.

Problem 2: Modules `dot2m`, `dot4m`, and `dot6m`, each have three inputs, `si`, `a` and `b`, and one output `dp`, and parameter `w`. In all of these modules output `dp` and input `si` are `w`-bit values. Inputs `a` and `b` are arrays with the number of elements matching the digit in the module name (two for `dot2m`, four for `dot4m`, etc). The inputs and outputs are (to be interpreted as) unsigned integers. When complete the output `dp` is set to the dot product of vectors `a` and `b` plus the scalar `si` (the computation is shown in the comments). Like those in Problem 1, these modules compute a dot product, but they have an extra input, `si`, which is added to the dot product. That extra input will come in handy when, say, using two `dot2m` modules in a `dot4m` module. (It will turn out that the Problem 1 dot product modules are better than the modules in this problem, and the disadvantage is more than just the need to do one extra addition.)

(*a*) Shown below is a completed multiply-add module and an incomplete `dot2m` module. Complete `dot2m` using instantiations of `madd`. As in Problem 1 do not use procedural code nor continuous assignments. See the checkbox items in the code for these and other requirements.

```
module madd
  #( int w = 8 )
   ( output uwire [w-1:0] s,  input uwire [w-1:0] si, a, b );
   assign s = si + a * b;
endmodule

module dot2m
  #( int w = 5 )
   ( output uwire [w-1:0] dp,
     input uwire [w-1:0] si, a[1:0], b[1:0] );

   // Compute:
   //   dp = si + a[0] * b[0] + a[1] * b[1];
   // Using instantiations of:
   //   madd.

endmodule
```

(*b*) Complete modules `dot4m` and `dot6m`. Consider using instantiations of `madd`, `dot2m`, and `dot4m`. (Don't instantiate `dot4m` in `dot4m`, we'll get to recursive instantiation soon enough.) As before, instantiate the minimum number of modules, and no procedural code and no continuous assignments can be used. See the checkbox items in the code for these and other requirements.

Problem 3: Modules `dot2y` and `dot4y` each have an output `dp` and inputs `a` and `b`, and two parameters `wa` and `wb`. Input `a` is an array of `wa`-bit elements and input `b` is an array of `wb`-bit elements. The number of bits in the output of `dot2y` is `wa + wb + 1`, which is the minimum number of bits needed to avoid overflow. (The number of bits in the product of a w_a-bit unsigned integer with a w_b-bit unsigned integer is $w_a + w_b$. If two w-bit unsigned integers are added the sum will require no more than $w + 1$ bits.) Similarly, `dp` in `dot4y` is `wa + wb + 2` bits.

Why go to the trouble of making the elements of `a` and `b` different sizes? To reduce cost and improve performance. In some applications, most notably neural networks, a large number of dot products (a component of matrix/vector multiplication) need to be computed. Furthermore, in neural network applications the precision of the operands can be made much lower than operands used in other application areas needing dot products, such as scientific computation. For that reason, specialized hardware for neural network computation often use lower-precision and mixed-precision arithmetic units.

(*a*) Complete `dot2y` using instantiations of modules `multy` and `addy`. Unlike the previous problems, here the connections of `multy` and `addy` will need to be modified. Consider the unsolved modules:

```
module multy
  #( int w = 5 )
   ( output uwire [w-1:0] p,
     input uwire [w-1:0] a,
     input uwire [w-1:0] b );
   // Modify the connections to this module.  (The stuff above this line.)
   assign p = a * b;
endmodule

module addy
  #( int w = 3 )
   ( output uwire [w-1:0] s,
     input uwire [w-1:0] a,
     input uwire [w-1:0] b );
   // Modify the connections to this module. (The stuff above this line.)
   assign s = a + b;
endmodule

module dot2y
  #( int wa = 5, wb = 6, wo = wa + wb + 1 )
   ( output uwire [wo-1:0] dp,
     input uwire [wa-1:0] a[1:0],
     input uwire [wb-1:0] b[1:0] );

   // Compute:
   //   dp = a[0] * b[0] + a[1] * b[1];
   // Using instantiations of:
   //   multy, addy

endmodule
```

To compute `a[0] * b[0]` needed for `dot2y` a multiply module is needed in which the two inputs and the output can each be different sizes. Modify `multy` so that it can be used in `dot2y`. Do the same for `addy`, then complete `dot2y` using these modules. Modify `multy` and `addy` so that they

can also be used for `dot4y`, if needed.

Make sure that the sizes of the ports are the minimum size needed. That is, don't make the number of bits in the multiplier output more than the sum of the bits in the two inputs. See the checkbox items in the code for these and other requirements.

(*b*) Complete `dot4y`. Consider instantiations of `dot2y`, `multy`, and `addy`. Take care to set parameter values so that the minimum number of bits are used in the ports. That is, when instantiating `dot2y` to make `dot4y` don't set the third `dot2y` parameter to something larger than `wa+wb+1`. (There is no need to set the third parameter at all.) See the checkbox items in the code for these and other requirements.

Verify that code is synthesizable by running the synthesis script. 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
dot2_w4               13288    1.89   100.0 ns      5 s
dot2m_w4              15482    2.45   100.0 ns      1 s
dot3_w4               20154    2.20   100.0 ns      2 s
dot4_w4               28021    2.80   100.0 ns      2 s
dot4m_w4              29645    2.66   100.0 ns      2 s
dot6m_w4              44379    3.67   100.0 ns      3 s
dot2_w10              87617    4.21   100.0 ns      6 s
dot2m_w10             94012    4.31   100.0 ns      6 s
dot3_w10             132035    4.67   100.0 ns      8 s
dot4_w10             177361    4.72   100.0 ns     10 s
dot4m_w10            183139    4.83   100.0 ns     11 s
dot6m_w10            272976    6.01   100.0 ns     16 s
dot2y_wa4_wb2         12180    2.51   100.0 ns      2 s
dot2y_wa6_wb4         40359    4.42   100.0 ns      4 s
dot4y_wa4_wb2         27229    3.46   100.0 ns      3 s
dot4y_wa6_wb4         85180    5.40   100.0 ns      7 s
```

LSU EE 4755 Homework 2 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`.

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.

(*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.

(*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.

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

(*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.

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.

(*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.

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.
- [ ] Run synthesis to find out which really is.

LSU EE 4755 Homework 3 Due: 20 Oct 2024 (evening)

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/hw03.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, `hw03.v`. Do this early enough so that minor problems (*e.g.*, password doesn't work) are minor problems.

Homework Overview

In string `((I am balanced))` the parentheses are balanced, but in `((Not balanced)` they are not because a closing parenthesis is missing and so an opening parenthesis is unmatched. The modules in this assignment examine a string containing parentheses and report the number of unmatched closing, `)`, and opening, `(`, parentheses. Both modules put these numbers on outputs `left_out_n_unmat_close` and `right_out_n_unmat_open`. The parentheses in `a(b)c` and `(()a)()` are correctly matched. The parentheses in `())`, `((())`, and `))((` are not. For inputs like `()`, `(())`, and `()()` both outputs should be zero. For `)` and `())` the number of unmatched closing parentheses is one and so output `left_out_n_unmat_close` should be `1` and output `right_out_n_unmat_open` should be `0`. See the testbench output for more examples.

In both modules, `pmatch_base` (Problem 1) and `pmatch_mark` (Problem 2), the string appears on input `str`. In `pmatch_mark` there is an additional output, `str_marked`, which should be set to a version of the string in which the correctly matched parentheses are replaced by angle brackets. For example, for input `str='(()'` the output shold be `str_marked='(<>'`. See the testbench output for more examples.

The modules each have parameter `n`. Input `str` and output `str_marked` are `n`-element unpacked (ordinary) arrays of 4-bit quantities. For convenience enumeration constants are defined for the characters used in this assignment:

```
typedef enum logic [3:0]
  { Char_Blank, Char_Dot,
    Char_Open, Char_Close,
    Char_Open_Okay, Char_Close_Okay } Char;
```

The input `str` can consist of any of the first four values. There is no distinction between `Char_Blank` and `Char_Dot`, they are stand-ins for arbitrary characters and neither affects paren-

thesis matching. The last two, `Char_Open_Okay` and `Char_Close_Okay` are to be used in Problem 2 for replacing properly matched parenthesis.

Those who are unsure of how to work with `str` or of just what the modules are supposed to do should examine modules `pmatch_comb_base` and `pmatch_comb_mark`. Module `pmatch_comb_base` will pass the testbench for Problem 1 (if it were renamed `pmatch_base`) and `pmatch_comb_mark` would pass the testbench for Problem 2 (if renamed).

These `comb` modules compute their results by scanning the string from left to right. In their synthesized hardware the critical path appears to be proportional to n, the string length. That's too long a critical path for our purposes. In Problems 1 and 2 this is to be overcome by using a recursive module structure that describes tree-like hardware. In a correct solution the critical path will be closer to $\log n$, and the cost will be lower too.

The synthesis output below shows the result of synthesizing the `comb` base module and a correct solution to Problem 1 at exponentially increasing string lengths ($n = 4,\ 8,\ 16,\ 32$). Notice that in the `comb` version the delay too increases exponentially (in proportion to n) while the delay in the Problem 1 solution increases more linearly. Absolute costs are lower too. The differences are less stark with a larger delay target. The default synthesis script uses the larger delay target to save time.

Module Name	Area	Delay Actual	Delay Target	Synth Time
pmatch_comb_base_n4_13	43915	0.82	0.1 ns	24 s
pmatch_comb_base_n8_29	175285	2.31	0.1 ns	185 s
pmatch_comb_base_n16_61	221959	6.99	0.1 ns	256 s
pmatch_comb_base_n32_125	771830	15.69	0.1 ns	772 s
pmatch_base_n4	22979	1.22	0.1 ns	15 s
pmatch_base_n8	73381	1.93	0.1 ns	50 s
pmatch_base_n16	142421	3.24	0.1 ns	86 s
pmatch_base_n32	341921	4.49	0.1 ns	179 s
pmatch_comb_base_n4	11039	2.14	900.0 ns	8 s
pmatch_comb_base_n8	33748	7.11	900.0 ns	77 s
pmatch_comb_base_n16	93278	18.98	900.0 ns	53 s
pmatch_comb_base_n32	248862	48.57	900.0 ns	117 s
pmatch_base_n4	15550	3.10	900.0 ns	6 s
pmatch_base_n8	41187	5.99	900.0 ns	5 s
pmatch_base_n16	98336	9.91	900.0 ns	18 s
pmatch_base_n32	216143	14.60	900.0 ns	23 s

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 instantiations of modules `pmatch_base` and `pmatch_mark`. The instantiations differ in the length of the string. At the end of execution the number of errors for each module at each size are shown. The output below is from a correctly solved assignment:

```
Total pmatch_base n=4: Errors: 0 cl, 0 op, 0 mk.
Total pmatch_base n=5: Errors: 0 cl, 0 op, 0 mk.
```

```
Total pmatch_base n=7: Errors: 0 cl, 0 op, 0 mk.
Total pmatch_base n=8: Errors: 0 cl, 0 op, 0 mk.
Total pmatch_base n=9: Errors: 0 cl, 0 op, 0 mk.
Total pmatch_base n=17: Errors: 0 cl, 0 op, 0 mk.
Total pmatch_mark n=4: Errors: 0 cl, 0 op, 0 mk.
Total pmatch_mark n=5: Errors: 0 cl, 0 op, 0 mk.
Total pmatch_mark n=7: Errors: 0 cl, 0 op, 0 mk.
Total pmatch_mark n=8: Errors: 0 cl, 0 op, 0 mk.
Total pmatch_mark n=9: Errors: 0 cl, 0 op, 0 mk.
Total pmatch_mark n=17: Errors: 0 cl, 0 op, 0 mk.
```

Each line starting with `Total` shows a tally of results. After `Total` the line shows the module name, either `pmatch_base` or `pmatch_mark`, and `n`, the length of the string. A tally of each output's error is shown after `Errors:`, `cl` is the number of incorrect closing-parentheses values, `op` is the number of opening-parenthesis values, and `mk` is the number of incorrectly marked strings.

Further up, the testbench shows sample output and error details. For each instantiation the first `n_errors_show = 5` incorrect outputs are shown on lines starting with `Error`. If it would help to see more then feel free to search for `n_errors_show` and increase the value. In addition the details of the first `n_samples_show = 6` correct outputs are shown on lines starting with `Sample`. The output below shows correct outputs:

```
Starting pmatch_base tests for n=5.
Sample pmatch_base n=5 '()   ': close = 0,  open = 0 (both correct)
Sample pmatch_base n=5 '.( ) ': close = 0,  open = 0 (both correct)
Sample pmatch_base n=5 ')(   ': close = 1,  open = 1 (both correct)
Sample pmatch_base n=5 ')    ': close = 1,  open = 0 (both correct)
Sample pmatch_base n=5 '))   ': close = 2,  open = 0 (both correct)
Sample pmatch_base n=5 '()))' : close = 2,  open = 0 (both correct)
Sample pmatch_base n=5 '())( ': close = 1,  open = 1 (both correct)
Sample pmatch_base n=5 '())((': close = 1,  open = 2 (both correct)
Sample pmatch_base n=5 '))(( ': close = 2,  open = 2 (both correct)
```

In the sample above the first `Sample` line indicates that for input `()` both the `left_out_n_unmat_close` and `right_out_n_unmat_open` outputs were 0, which is correct because there were no unmatched parentheses. The second sample is also properly matched. It consists of a dot (which is just an ordinary character), parentheses, and spaces. The third sample has both one unmatched opening parenthesis and an unmatched closing parenthesis.

The testbench starts applying patterns found in `str_special`. Feel free to add your own to help with debugging. After the patterns in `str_special` are used the testbench will make up random patterns.

The output below is of a run using a partially correct `pmatch_base`:

```
Sample pmatch_base n=5 '()   ': close = 0,  open = 0 (both correct)
Sample pmatch_base n=5 '.( ) ': close = 0,  open = 0 (both correct)
Sample pmatch_base n=5 ')(   ': close = 1,  open = 1 (both correct)
Sample pmatch_base n=5 ')    ': close = 1,  open = 0 (both correct)
Error pmatch_base n=5 '))   ': close  0 != 2 (correct)
Sample pmatch_base n=5 '()))' : close = 2,  open = 0 (both correct)
Error pmatch_base n=5 '))(( ': close  0 != 2 (correct)
Error pmatch_base n=5 '))   ': close  0 != 2 (correct)
```

The errors reported above seem to show that the `left_out_n_unmat_close` is wrong when the string starts with two consecutive closing parentheses.

The `str_marked` output of module `pmatch_mark` is supposed to show the string with the correctly matched parentheses replaced by angle brackets (actually less-than and greater-than symbols). Appearing below is sample output of the module in a correctly solved assignment. Two lines are used to show the result of each input. The first shows the input string, such as `()` in the first sample, the second line shows the marked string, such as `<>`.

```
Sample pmatch_mark n=5 '()   ': close = 0,  open = 0 (both correct)
Sample pmatch_mark n=5 '<>   ' (marked_outpuut)
Sample pmatch_mark n=5 '.( ) ': close = 0,  open = 0 (both correct)
Sample pmatch_mark n=5 '.< > ' (marked_outpuut)
Sample pmatch_mark n=5 ')(   ': close = 1,  open = 1 (both correct)
Sample pmatch_mark n=5 ')(   ' (marked_outpuut)
Sample pmatch_mark n=5 ')    ': close = 1,  open = 0 (both correct)
Sample pmatch_mark n=5 ')    ' (marked_outpuut)
Sample pmatch_mark n=5 '))   ': close = 2,  open = 0 (both correct)
Sample pmatch_mark n=5 '))   ' (marked_outpuut)
Sample pmatch_mark n=5 '()))' ': close = 2,  open = 0 (both correct)
Sample pmatch_mark n=5 '<>)) ' (marked_outpuut)
Sample pmatch_mark n=5 '())( ': close = 1,  open = 1 (both correct)
Sample pmatch_mark n=5 '<>)( ' (marked_outpuut)
Sample pmatch_mark n=5 '())((': close = 1,  open = 2 (both correct)
Sample pmatch_mark n=5 '<>)((' (marked_outpuut)
Sample pmatch_mark n=5 '))(( ': close = 2,  open = 2 (both correct)
Sample pmatch_mark n=5 '))(( ' (marked_outpuut)
```

The output below is of a run using an incorrect `pmatch_mark` module. An error line is printed for each incorrect output, `left_out_n_unmat_close`, `right_out_n_unmat_open`, and `str_marked`. For a particular input, say `()`, a module can have one incorrect output, such as `left_out_n_unmat_close`, while the other two outputs, `right_out_n_unmat_open` and `str_marked` are correct. That's the case in the first and last error below:

```
Starting pmatch_mark tests for n=5.
Error pmatch_mark n=5 '()   ': close  1 != 0 (correct)
Error pmatch_mark n=5: '<<>>)' != '<>   ' (correct)
Error pmatch_mark n=5 '.( ) ': open   3 != 0 (correct)
Error pmatch_mark n=5: '..((' != '.< > ' (correct)
Error pmatch_mark n=5 ')(   ': close  2 != 1 (correct)
```

References and Helpful Examples

The modules in this assignment must be recursively defined, so that they describe a tree-like structure. See the CLZ module from 2019 Homework 2. The assignment, solution, and live version done in class are part of the 2024 assignment directory, look for the file names starting `2019`. Understanding the `clz_tree_fat` solution is sufficient. The trick used to avoid the adder in the clz module is not relevant to this 2024 assignment.

Problem 1: Module `pmatch_base` has one input, `str`, and two outputs `left_out_n_unmat_close` and `right_out_n_unmat_open`, and parameters `n` and `wn`. Input `str` is an `n`-element array of 4-bit quantities called characters, with `str[0]` being the leftmost character and `str[n-1]` being the rightmost character. Outputs `left_out_n_unmat_close` and `right_out_n_unmat_open` are each `wn` bits. They should be set to the number of unmatched parentheses as described in the introduction to the assignment represented as an unsigned integer. The default value of `wn`, $\lceil \lg(n+1) \rceil$, is the minimum value needed to correctly report `n` mismatched parentheses. (Setting `wn` to a larger values is a potential waste.)

Complete the module so that it produces these outputs and so that it describes tree-structured hardware by using recursion. The critical path should be proportional to $\log n$, which can be achieved by splitting the `str` input between two recursive instantiations and combining their outputs. It may help to examine `pmatch_comb_base`, which produces the same outputs, though not recursively.

- ☐ Complete the module so that the testbench reports zero errors.
- ☐ The module description must be recursive and describe tree-like hardware.
- ☐ Set `wn` to the smallest correct value in the recursive instantiations.
- ☐ Make sure the module is synthesizable using command `genus -files syn.tcl`.

Hint: Consider checking `str` *only in the base (terminal) case of the recursion.*

Problem 2: Module `pmatch_mark` has three inputs, `str`, `left_in_n_unmat_open`, `right_in_n_unmat_close`, and three outputs `left_out_n_unmat_close`, `right_out_n_unmat_open`, `str_marked`, and parameter `n`. Input `str` and output `str_marked` are `n`-element 4-bit arrays. Input `str` carries a string, and output `str_marked` is to be set to a version of the input string with each properly matched `Char_Open` replaced with a `Char_Open_Okay` and each properly matched `Char_Close` replaced with a `Char_Close_Okay`.

Input `left_in_n_unmat_open` is set to the number of unmatched opening parentheses to the left of `str`. For example, suppose `str='))'` and `left_in_n_unmat_open=1`. Then that means that one of the unmatched parentheses in `str` is matched by something to the left of `str`. For this example, the value of `str_marked='>)'` because `left_in_n_unmat_open=1`. Similarly input `right_in_n_unmat_close` is set to the number of unmatched closing parentheses to the right of the string.

The testbench will always set `left_in_n_unmat_open` and `right_in_n_unmat_close` to zero. Your module should set them correctly in connections to recursive instantiations so that they can determine which of their parentheses are matched.

Outputs `left_out_n_unmat_close`, `right_out_n_unmat_open` should have the same values as they would in Problem 1. That is, they should show the number of unmatched opening and closing parentheses in `str` *even if those parentheses are marked as matched due to values of* `left_in_n_unmat_open` *and* `right_in_n_unmat_close`. For example, suppose `str='))))()('` and `left_in_n_unmat_open=2` and `right_in_n_unmat_close=1`. The module should set output `left_in_n_unmat_open=3` (ignoring the 2 matches) and `right_in_n_unmat_close=1` (ignoring the one match), and set output `str_marked='>>)<><'`, showing the matched parentheses. Parentheses are set as matched both when they are matched by parentheses within `str` and when they are matched by parentheses reported by `left_in_n_unmat_open` and `right_in_n_unmat_close`.

- ☐ Complete the module so that the testbench reports zero errors.
- ☐ The module description must be recursive and describe tree-like hardware.

- [ ] Set `wn` to the smallest correct value in the recursive instantiations.
- [ ] Make sure the module is synthesizable using command `genus -files syn.tcl`.

Hint: Consider writing `str_marked` *only in the base (terminal) case of the recursion.*

LSU EE 4755 Homework 4 Due: 8 Nov 2024

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 the students' responsibility 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, 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**.

Problems start on next page.

Problem 1: Appearing below is the base case from module `pmatch_mark` in the solution to Homework 3 Problem 2.

```
typedef enum logic [3:0]
  { Char_Blank = 0,     Char_Dot = 1,
    Char_Open = 2,      Char_Close = 3,
    Char_Open_Okay = 4, Char_Close_Okay = 5 } Char;

module pmatch_mark
  #( int n = 5, wn = $clog2(n+1) )
   ( output logic [wn-1:0] left_out_n_unmat_close, right_out_n_unmat_open,
     output uwire [3:0] str_marked [0:n-1],
     input uwire [wn-1:0] left_in_n_unmat_open, right_in_n_unmat_close,
     input uwire [3:0] str [0:n-1] );

   if ( n == 1 ) begin

      assign left_out_n_unmat_close = str[0] == Char_Close;
      assign right_out_n_unmat_open = str[0] == Char_Open;

      assign str_marked[0] =
             str[0] == Char_Close && left_in_n_unmat_open   ? Char_Close_Okay :
             str[0] == Char_Open  && right_in_n_unmat_close ? Char_Open_Okay  :
             str[0];
```

☐ Show the hardware that will be inferred for the base case (`n==1`) shown above.

☐ Show the hardware after optimization and ☐ for the default value of `wn`.

☐ In the optimized hardware do not show comparison units, instead show the individual gates performing the comparison, ☐ optimizing for constant values.

Problem 2: Appearing below are three variations on a module that will set its output to either the input value, or a maximum value if the input is larger. The module will always be instantiated with `wl < wn`. All of them are functionally equivalent, but were synthesized to different costs (by Genus 23.12-s086.1 when similar code was used in the solution to Homework 3). Because they are functionally equivalent a perfect synthesis program would synthesize each to the same hardware (with equal costs).

```
module clamp_plan_a
  #( int wl = 3, wn = 4 ) ( output uwire [wl-1:0] x, input uwire [wn-1:0] a );
   localparam logic [wl-1:0] nl_max = ~(wl)'(0); // Sequence of wl 1s.
   assign x = a <= nl_max ? a : nl_max;
endmodule

module clamp_plan_b
  #( int wl = 3, wn = 4 ) ( output uwire [wl-1:0] x, input uwire [wn-1:0] a );
   localparam logic [wl-1:0] nl_max = ~(wl)'(0); // Sequence of wl 1s.
   assign x = a < nl_max ? a : nl_max;
endmodule

module clamp_plan_c
  #( int wl = 3, wn = 4 ) ( output uwire [wl-1:0] x, input uwire [wn-1:0] a );
   localparam logic [wl-1:0] nl_max = ~(wl)'(0); // Sequence of wl 1s.
   assign x = !a[wn-1:wl] ? a : nl_max;
endmodule
```

☐ Show the optimized hardware for the low-cost version(s).

☐ Find the simple-model cost of each after optimization. The costs should be ☐ in terms of `wn` and `wl`.

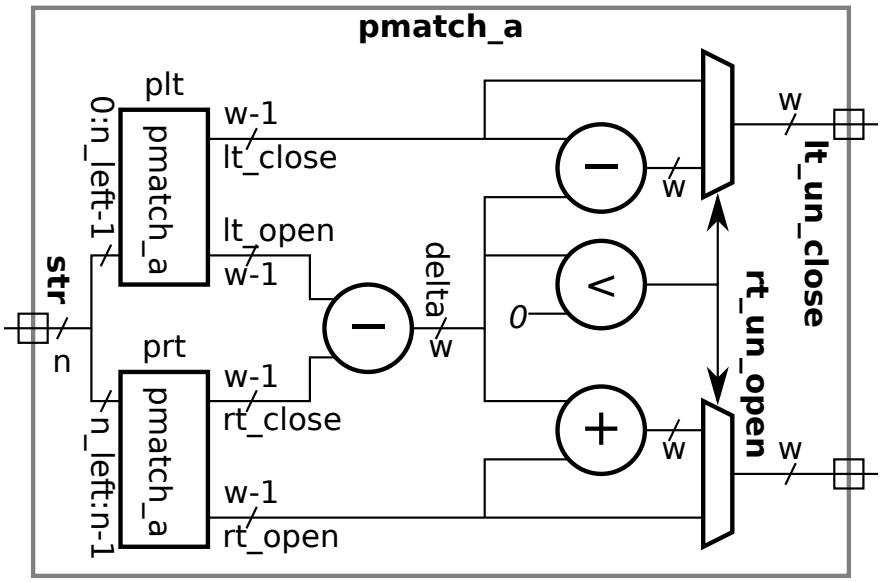
☐ Find the simple-model delay of each after optimization. The delays should be ☐ in terms of `wn` and `wl`.

Problem 3: Appearing on the next page is a simplified solution to Homework 3, Problem 2. In this module the number of bits in the connections carrying parentheses counts is hardcoded to 8. Though the hardware is correct for $n < 256$ it is more costly and slower than it needs to be. But for this problem it's good enough.

Show the Homework 3, Problem 2 hardware that will be inferred for this module for $n > 1$ (the non-base case). That is, don't show the hardware computing `left_out_n_unmat_close` and `right_out_n_unmat_open`.

- ☐ Show the inferred hardware at one level for $n > 1$.
- ☐ Feel free to use abbreviations.
- ☐ Don't show the Homework 3 Problem 1 hardware (the last `always_comb`).
- ☐ Don't confuse elaboration-time computation with hardware.

For reference, the hardware for the Homework 3 Problem 1 part of this module is shown below.



```
module pmatch_mark_big #( int n = 5, wn = 8 )
   ( output logic [wn-1:0] left_out_n_unmat_close, right_out_n_unmat_open,
     output uwire [3:0] str_marked [0:n-1],
     input uwire [wn-1:0] left_in_n_unmat_open, right_in_n_unmat_close,
     input uwire [3:0] str [0:n-1] );

   if ( n == 1 ) begin
      assign left_out_n_unmat_close = str[0] == Char_Close;
      assign right_out_n_unmat_open = str[0] == Char_Open;
      assign str_marked[0] =
             str[0] == Char_Close && left_in_n_unmat_open   ? Char_Close_Okay :
             str[0] == Char_Open  && right_in_n_unmat_close ? Char_Open_Okay  : str[0];
   end else begin

      localparam int n_left = n/2,   n_right = n - n_left;
      localparam int wl = 8,         wr = 8;  // Note: this is wasteful.

      uwire [wl-1:0] lt_close, lt_open;
      uwire [wr-1:0] rt_close, rt_open;
      logic [wl-1:0] lt_matched_op, lt_matched_cl;
      logic [wr-1:0] rt_matched_op, rt_matched_cl;

      pmatch_mark_big #(n_left,wl) plt   // Recursive Instantiation
        ( lt_close, lt_open, str_marked[0:n_left-1],
          lt_matched_cl, lt_matched_op, str[0:n_left-1] );
      pmatch_mark_big #(n_right,wr) prt  // Recursive Instantiation
        ( rt_close, rt_open, str_marked[n_left:n-1],
          rt_matched_cl, rt_matched_op, str[n_left:n-1] );

      always_comb begin
         logic signed [wn-1:0] more_op, more_cl;

         lt_matched_cl = left_in_n_unmat_open;
         rt_matched_op = right_in_n_unmat_close;

         more_op = left_in_n_unmat_open - lt_close;
         rt_matched_cl = more_op < 0 ? lt_open : more_op + lt_open;

         more_cl = right_in_n_unmat_close - rt_open;
         lt_matched_op = more_cl < 0 ? rt_close : more_cl + rt_close;
      end

      always_comb begin          // Same as Homework 3 Problem 1
         logic signed [wn-1:0] delta;
         delta = lt_open - rt_close;
         left_out_n_unmat_close = delta >= 0 ? lt_close : lt_close - delta;
         right_out_n_unmat_open = delta < 0  ? rt_open  : rt_open + delta;
      end
   end
endmodule
```

LSU EE 4755 Homework 5 Due: 18 Nov 2024 (evening)

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/hw05.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 the students' responsibility 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, 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 somehow necessary at this point), copy the assignment, and run the Verilog simulator on the unmodified homework file, `hw05.v`. Do this early enough so that minor problems (*e.g.*, password doesn't work) are minor problems.

Homework Overview

The module in this assignment, `dot_seq_2`, computes a dot product of two vectors, `a` and `b`, each of length n, where n is even (and greater than zero). At each cycle two elements of `a` and `b` arrive, so it takes $n/2$ cycles for the complete vectors to arrive. The connections to `dot_seq_2` are:

```
module dot_seq_2 #( int w = 5, wi = 4 )
   ( output logic [w-1:0] dp,
     output logic [wi-1:0] first_id, last_id,
     input uwire [w-1:0] a[2], b[2],
     input uwire [wi-1:0] in_id,
     input uwire reset, first, last,
     input uwire clk );
```

Each element of input vector `a` and `b` is `w` bits. Inputs `a` and `b` carry the first two elements of each vector when input `first=1`. At each subsequent cycle `a` and `b` will carry two more elements of the vector until reaching a cycle where `last=1`, in which case `a` and `b` carry the last two elements of each vector. (See the testbench section for examples.) If `first=1` and `last=1` in the same cycle

then the vectors are just of length 2. There is no upper bound on the vector length. Some time after `last=1` output `dp` is to be set to the dot product of the vectors.

In the unmodified assignment `dp` is set to the correct value in the same cycle that `last=1`. In a correct solution `dp` should be set three cycles later to limit the critical path (see the problem description).

Input `in_id` is a `wi`-bit ID number which is used to identify the beginning and end of the vector. Outputs `first_id` and `last_id` are each `wi`-bit ID numbers. In a correctly solved assignment `first_id` is the ID of the beginning of the vector whose dot product appears on `dp` and `last_id` is the ID appearing at the end of that vector. In the unmodified assignment `last_id` is set to the correct value (though along with `dp` it is set too soon).

When input `reset=1` any in-progress dot-product computation is abandoned that outputs `first_id` and `last_id` are set to zero.

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 `dot_seq_2` and report on the results. Unlike other assignments, only one module instantiation will be tested.

The testbench will show a trace for at least the first 20 cycles, with additional trace output shown before errors. The trace lines from a correctly solved assignment appear below.

```
 2 _F_ ID a8  A 01,01  B 01,10  exp: fid 00  lid 00  dp 00  MOD: FID 00  LID 00  DP xx
 3 ___ ID aa  A 01,01  B 01,10  exp: fid 00  lid 00  dp 00  MOD: FID 00  LID 00  DP xx
 4 ___ ID ad  A 01,01  B 01,10  exp: fid 00  lid 00  dp 00  MOD: FID 00  LID 00  DP xx
 5 ___ ID af  A 01,01  B 01,10  exp: fid 00  lid 00  dp 00  MOD: FID 00  LID 00  DP xx
 6 __L ID b1  A 01,01  B 01,10  exp: fid a8  lid b1  dp 55  MOD: FID 00  LID 00  DP xx
 7 ___ ID b3  A 01,01  B 01,10  exp: fid a8  lid b1  dp 55  MOD: FID 00  LID 00  DP xx
 8 _F_ ID b5  A 01,01  B 01,10  exp: fid a8  lid b1  dp 55  MOD: FID 00  LID 00  DP xx
 9 ___ ID b8  A 01,01  B 01,10  exp: fid a8  lid b1  dp 55  MOD: FID a8  LID b1  DP 55
10 ___ ID b9  A 01,01  B 01,10  exp: fid a8  lid b1  dp 55  MOD: FID a8  LID b1  DP 55
11 __L ID ba  A 01,01  B 01,10  exp: fid b5  lid ba  dp 44  MOD: FID a8  LID b1  DP 55
12 _FL ID bc  A 01,01  B 01,10  exp: fid bc  lid bc  dp 11  MOD: FID a8  LID b1  DP 55
13 ___ ID bd  A 01,01  B 01,10  exp: fid bc  lid bc  dp 11  MOD: FID a8  LID b1  DP 55
14 ___ ID c0  A 01,01  B 01,10  exp: fid bc  lid bc  dp 11  MOD: FID b5  LID ba  DP 44
15 ___ ID c3  A 01,01  B 01,10  exp: fid bc  lid bc  dp 11  MOD: FID bc  LID bc  DP 11
16 _F_ ID c4  A 01,01  B 01,10  exp: fid bc  lid bc  dp 11  MOD: FID bc  LID bc  DP 11
17 ___ ID c7  A 01,01  B 01,10  exp: fid bc  lid bc  dp 11  MOD: FID bc  LID bc  DP 11
18 ___ ID ca  A 01,01  B 01,10  exp: fid bc  lid bc  dp 11  MOD: FID bc  LID bc  DP 11
19 ___ ID cc  A 01,01  B 01,10  exp: fid bc  lid bc  dp 11  MOD: FID bc  LID bc  DP 11
```

Trace lines start with a cycle number, in the sample above they go from 2 to 19 as of this writing. That is followed by three characters that show the state of the `reset`, `first`, and `last` inputs (in that order). In cycle 2 `first=1` and the other two are zero. In cycle 12 `first=1` and `last=1`. In the sample above `reset` is always `0`.

Upper-case labels, such as `ID` and `FID` are used for module inputs and outputs. Lower-case labels, such as `lid` and `dp` are used to show what the testbench expects to find at the outputs (though the expectation can be for several cycles later). Label `exp:` is an abbreviation for expected outputs, and label `MOD:` indicates module outputs. (A reminder of what the lower- and upper-case labels signify.)

Each `a` and `b` input has two values. Both values appear next to `A` and `B`, shown in hexadecimal. In the first tests the `a` and `b` vectors are set to make it easy to debug problems: each component of vector `a` is 1. The even components of `b` are 1 and the odd components of `b` are set to $16_{10} = 10_{16}$.

Consider the trace for cycle 9. The `dp` output is 55 and the IDs are `a8` (first) and `b1` (last). The vectors started arriving in cycle 2 (note the matching `a8` on the module input, labeled `ID`), and the last pair of elements arrived at cycle 6. Though the vector finished arriving at cycle 6, its dot product did not appear at the module output until cycle 9. Also consider cycle 8. A new vector starts arriving, as one can tell by the `first=1` signal. The newly arriving vector does not interfere with the computation of the vector that finished arriving in cycle 6. Dot products for both will be computed.

The testbench will print a summary of results at the end, for the correct solution:

```
Done with 10000 tests. 0 dp errs ( 9718 correct )
Done with 10000 tests. 0 FID errs, 0 LID errs.
Done with 10000 tests. Correct 9718, avg latency 3.0 cyc Okay
```

The first line above indicates that there were zero errors with the `dp` output. A total of 10000 vectors were offered to the module and the module finished 9718 of them. The $10000 - 9718 = 282$ other vectors were interrupted by the `reset` signal. In the second line `FID errs` indicates how many `first_id` outputs were incorrect, the same for `LID errs`. The last line indicates the average number of cycles to compute a dot product. In this assignment that's expected to be an integer, and equal to 3.

When there are errors the trace indicates which outputs are wrong by changing labels `FID`, `LID`, and `DP` to `ERR` if the respective output value is incorrect. The expected correct values of `first_ID` and `last_ID` are based on the `dp` output. For example, if `first_ID` and `last_ID` are changed for a new vector but `dp` shows the dot product of the previous vector, then the values of `first_ID` and `last_ID` will be considered wrong and so `FID` and `LID` will show `ERR`.

If the `dp` value is correct but arrives in less than 3 cycles an error message will be printed. Also, the `DP` label will be changed to `EY` (early) for a correct dot product value that appears earlier than 3 cycles after `last` arrived.

Note that the `exp` (lower case) labels show values that are expected now or in the future. For example, in cycle 6 below the `dp` label shows a 55 based on the arrival of the `last` signal. However, that `dp` value should not be seen at the outputs until 3 cycles later.

Also additional detail is provided in the first line following an error. Consider the output from the unmodified assignment:

```
   2 _F_ ID a8  A 01,01  B 01,10  exp: fid 00  lid 00  dp 00  MOD: FID 00  LID 00  EY 11
   3 ___ ID aa  A 01,01  B 01,10  exp: fid 00  lid 00  dp 00  MOD: FID 00  LID 00  DP 22
   4 ___ ID ad  A 01,01  B 01,10  exp: fid 00  lid 00  dp 00  MOD: FID 00  LID 00  DP 33
   5 ___ ID af  A 01,01  B 01,10  exp: fid 00  lid 00  dp 00  MOD: FID 00  LID 00  DP 44
   6 __L ID b1  A 01,01  B 01,10  exp: fid a8  lid b1  dp 55  MOD: ERR 00  LID b1  EY 55
Error first ID: 0 != a8 (correct)
Error dp timing: latency 0 cyc < 3 cyc (minimum)
   7 ___ ID b3  A 01,01  B 01,10  exp: fid a8  lid b1  dp 55  MOD: FID 00  LID b1  ER 66
Error dp: 66 != 55 (correct)
```

At cycle 6 output `dp` is set to the correct value, 55 (see the `dp` item) as is the `LID`, but since dot product arrives early the label is `EY` rather than `DP`. Also, the `first_id` output is incorrect, 0, so it is labeled `ERR` in the trace and the correct value is shown, `a8`. At cycle 7 the `DP` output is incorrect, it should have stayed at `55`.

Helpful Examples

A pipelined execution unit was the subject of 2016 Homework 6. Other than the need to do

floating-point computation, the 2016 Homework 6 pipeline was simpler than the one in this assignment. Another pipelined calculation (something to be covered in class on on Friday 15 November 2024) is the subject of 2021 Homework 6. Past assignments and their solutions are linked to the assignments and exams page.

Problem 1: In the unsolved assignment `dot_seq_2` computes the correct dot products, but does not accompany those with the correct `first_id` value, and the synthesized module would have a longer critical path then desired. Complete the module so that `first_id` is set to the correct ID and meeting the following timing requirements:

- ☐ Don't perform any computation on values arriving at inputs. Instead, write them to registers and start computation in the next cycle.
- ☐ The critical path can include at most one arithmetic operation on the vector components. For example, don't compute `a[0]*b[0]+a[1]*b[1]` because that operation has a critical path of 2 operations: a multiply and add. Instead compute the products on one cycle and the sum in another.
- ☐ The correct `dp` should be available three cycles after the `last=1`.
- ☐ Once the `dp`, `first_id`, and `last_id` outputs are set they should remain unchanged until either the `reset` is asserted or until the next dot product is ready.
- ☐ The module must be synthesizable. Verify with the command `genus -files syn.tcl`.

LSU EE 4755 Homework 6 Due: 1 Dec 2024

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, 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**.

Problems start on next page.

Problem 1: Show the hardware that will be synthesized for the posted solution to Homework 5. The solution (with fewer comments than the posted version) is shown below.

```
module dot_seq_2 #( int w = 5, wi = 4 )
   ( output logic [w-1:0] dp,          output logic [wi-1:0] first_id, last_id,
     input uwire [w-1:0] a[2], b[2],   input uwire [wi-1:0] in_id,
     input uwire reset, first, last,   input uwire clk );

   logic [w-1:0] pl_a[1:1][2], pl_b[1:1][2];  // Arriving vector elements.
   logic [w-1:0] pl_prod[2:2][2];             // Vector products.
   logic [w-1:0] pl_sum[3:3];                 // Dot prod of 2-element segment.
   logic [wi-1:0] pl_id[1:3];                 // ID.
   logic [1:0] pl_fl[1:3];                    // The first and last signals.
   logic [wi-1:0] acc_id;
   logic [w-1:0] acc_sum;

   always_ff @( posedge clk ) begin
      /// Stage 0
      pl_a[1] <= a;  // This copies both elements of a.
      pl_b[1] <= b;
      pl_id[1] <= in_id;
      pl_fl[1] <= reset ? 2'b0 : {last,first};

      /// Stage 1
      for ( int i=0; i<2; i++ ) pl_prod[2][i] <= pl_a[1][i] * pl_b[1][i];
      pl_id[2] <= pl_id[1];
      pl_fl[2] <= reset ? 2'd0 : pl_fl[1];

      /// Stage 2
      pl_sum[3] <= pl_prod[2][0] + pl_prod[2][1];
      pl_id[3] <= pl_id[2];
      pl_fl[3] <= reset ? 2'h0 : pl_fl[2];

      /// Stage 3
      begin
         automatic logic s3_first = pl_fl[3][0]; // For readability.
         automatic logic s3_last =  pl_fl[3][1]; // For readability.
         automatic logic [w-1:0] s3_sum = s3_first ? pl_sum[3] : pl_sum[3] + acc_sum;

         acc_sum <= s3_sum;

         if ( reset ) begin
            first_id <= 0;
            last_id <= 0;
         end else begin
            if ( s3_first ) acc_id <= pl_id[3];
            if ( s3_last ) begin
               first_id <= s3_first ? pl_id[3] : acc_id;
               last_id <= pl_id[3];
               dp <= s3_sum;
            end
         end
      end
   end
endmodule
```

Problem 2: Solve 2023 Final Exam Problem 2, which asks for a cost and delay analysis of a word count module.

2 Fall 2023

LSU EE 4755 Homework 1 Due: 7 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/hw01.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, copy the assignment, and run the Verilog simulator on the unmodified homework file, `hw01.v`. Do this early enough so that minor problems (*e.g.*, password doesn't work) are minor problems.

Testbench

To compile your code and run the testbench press F9 in an Emacs buffer in a properly set up account. The testbench will test 3 modules, `minmax2p1` ($n = 2$), `minmax4` ($n = 4$), and `minmax8` ($n = 8$). Each module will be tested on 100 inputs. If a module's output on a particular input is incorrect, a message will be printed showing the incorrect and correct output. This output will only be shown for the first few errors, but a tally will be shown near the end counting all errors.

In an unmodified assignment the testbench will generate output that includes the following near the end:

```
Error n=8  max    z != 8107 (correct)
Error n=8  min    z !=  907 (correct)
Error n=8  max    z != 8156 (correct)
Error n=8  min    z !=  243 (correct)
Error n=8  max    z != 6424 (correct)
Done with n=8, tests, 100 min 100 max errors found.
xmsim: *W,RNQUIE: Simulation is complete.
xcelium> exit
Total number of errors: 600
```

The `z` in the output above means that the `minmax8` outputs (both `min` and `max`) were set to `z`, meaning the output was not connected (z is often used to indicate high impedance). The output of the testbench for a correctly completed assignment is:

```
Done with n=2, tests, 0 min 0 max errors found.
Done with n=4, tests, 0 min 0 max errors found.
Done with n=8, tests, 0 min 0 max errors found.
```

```
xmsim: *W,RNQUIE: Simulation is complete.
xcelium> exit
Total number of errors: 0
```

Helpful Examples

A good past assignment to look at is 2017 Homework 1. A question very similar to Problem 1 was asked in 2018 Homework 1 Problem 1. So, do not look at 2018 Homework 1 until after you have made a very serious attempt at Problem 1.

Problem 1: Module `minmax2`, shown below, sets output `min` to the smaller of its two inputs `a0` and `a1`, and sets `max` to the larger of those two inputs:

```
module minmax2
  #( int w = 10 )
   ( output uwire [w-1:0] min, max,   input uwire [w-1:0] a0, a1 );
   assign { min, max } = a0 <= a1 ? { a0, a1 } : { a1, a0 };
endmodule
```

Notice that `minmax2` uses a continuous assignment statement. Complete module `minmax2p1` so that it does the same thing as `minmax2`, but without a continuous assignment and without procedural code. Instead instantiate `compare_lt` and `mux2` modules (shown below). Follow other guidelines and requirements shown in the checkboxes in the Verilog file.

```
module compare_lt
  #( int w = 31 )
   ( output uwire lt, input uwire [w-1:0] a0, a1 );

   // Set lt to 0 if a1 < a0, set lt to 1 otherwise.
   assign lt = a0 <= a1;
endmodule

module mux2
  #( int w = 3 )
   ( output uwire [w-1:0] x,   input uwire s,  input uwire [w-1:0] a0, a1 );
   assign x = s ? a1 : a0;
endmodule
```

There is another problem on the next page.

Problem 2: Modules `minmax4` and `minmax8` each have outputs `min` and `max`, which are to be set to the smallest and largest values of their input. Input `a` to `minmax4` is a 4-element array of `w`-bit unsigned integers, and input `a` to `minmax8` is an 8-element array. Complete these modules as described below.

For this problem use modules `minmax2`, `min2`, and `max2`. Module `min2`, as one might guess, sets its output to the smaller of its two inputs. Module `max2` is similar. Assume that the combined cost of a `min2` and `max2` module is greater than one `minmax2` (but less than the cost of two `minmax2` modules).

```
module min2 #( int w = 10 )
             ( output uwire [w-1:0] min, input uwire [w-1:0] a0, a1 );
   assign min = a0 < a1 ? a0 : a1;
endmodule

module max2 #( int w = 10 )
             ( output uwire [w-1:0] max, input uwire [w-1:0] a0, a1 );
   assign max = a0 < a1 ? a1 : a0;
endmodule
```

(*a*) Complete module `minmax4` using instantiations of modules `minmax2`, and possibly `min2` and `max2` as needed. Do not use `assign` statements or procedural code. Follow other guidelines shown in the checklist in the code. Pay attention to the relative cost of the `min2`, `max2`, and `minmax2` modules.

(*b*) Complete module `minmax8` using instantiations of modules `minmax4`, and possibly `minmax2`, `min2`, and `max2` as needed. Do not use `assign` statements or procedural code. Follow other guidelines shown in the checklist in the code. Pay attention to the relative cost of the `min2`, `max2`, and `minmax2` modules.

The code must by synthesizable. To synthesize your code issue the command `genus -files syn.tcl`. If there are no errors, running this command will generate output that includes like the following:

```
Module Name                          Area   Delay   Delay
                                           Actual  Target
minmax2p1_w8                        15086   2.191  10.000 ns
minmax2_w8                          15086   2.191  10.000 ns
minmax4_w8                          49094   4.412  10.000 ns
minmax8_w8                         117111   6.632  10.000 ns
minmax2p1_w8_1                      19678   1.029   0.100 ns
minmax2_w8_1                        19678   1.029   0.100 ns
minmax4_w8_1                        75084   1.496   0.100 ns
minmax8_w8_1                       214774   2.567   0.100 ns
```

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.)

LSU EE 4755 Homework 3 Due: 13 October 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/hw03.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, `hw03.v`.

Homework Overview

As we probably know a *permutation* is a rearrangement of distinct objects. If there are n objects there are $n!$ permutations, including the identity permutation (which leaves the objects in their original positions). For example, the three-letter sequence `abc` can be permuted 6 ways: `abc, acb, bac, bca, cab, cba`. (In module `perm` input `pdata_in` is an unpermuted sequence and output `pdata_out` should be set to a permutation of the input.) There are many ways to specify which permutation we want. We could just say, I want permutation `acb`, meaning leave the first element unchanged and swap the next two. So permutation `acb` of `xyz` would be `xzy`. When specifying permutations this way it is more common to use digits, so rather than `acb` we would say permutation `021` of `xyz`. Here `021` indicates how we want things rearranged and `xyz` are the objects before being re-arranged and `xzy` are after the rearrangement.

Suppose we want to generate all permutations in, say, a loop. We might have a current permutation, `021`, and would like to generate the next one, say `102` (or `bac`). One way of doing that is using a *factorial number*. (This was the subject of `https://xkcd.com/2835`.) A factorial number is a mixed-radix number. (In module `perm` input `pnum_in` and output `pnum_out` are both factorial numbers. In the testbench the factorial numbers are called indices.) In an n-digit factorial number digit 0 (the LSD) is radix 1, digit 1 is radix 2 (binary), digit 2 is radix 3, and so on, to digit $n-1$. Digit 0, by the way, being radix 1, must always be zero. Digit 1 can be 0 or 1, digit 2 can be 0,1,2, etc. When all digits are zero the number specifies an identity permutation. Digit i of a factorial number specifies where to get the value to put in position i of the permuted sequence. To see examples of factorial numbers and the respective permutations look at the sample testbench outputs below.

With factorial numbers it's easy to compute the next permutation in a sequence: just add one. Start at the least significant digit. If there is a carry out (and there always is at the least significant digit) proceed to the next digit. Denote the value of digit i (radix $i+1$) as $d_i \in [0, i]$. Adding a

carry in to the digit yields $d_i + 1$. If $d_i + 1 \leq i$ then that is the new value of of d_i and the carry out propagation stops. Otherwise the new value of d_i is zero and proceed to digit $i + 1$.

Code for computing the next permutation is shown in module `perm_behavioral`. That module also shows how to apply a factorial number (`pnum_in`) to permute items in `pdata_in` and connect them to `pdata_out`.

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 module `perm`. The instantiations differ on the number of items to permute, `n`, and the number of bits in each item, `w`. The testbench shows sample outputs and errors, and ends with a tally of errors for each instantiation. The output for an unmodified assignment includes:

```
Starting tests for w=8, n=3
Trace of permutation: 0 0 0  -> a b c
Error in next index:  0 0 0  -> 0 0 0   !=  0 1 0  (correct)
Error in permutation: 0 1 0  -> a b c   !=  a c b  (correct)
Error in next index:  0 1 0  -> 0 1 0   !=  1 0 0  (correct)
[snip]

Finished with n=10, 999 perm errors, 1000 next idx errors in 1000 tests.
End of tests n=3, 5 perm errors, 6 next idx errors for 6 tests.
End of tests n=4, 23 perm errors, 24 next idx errors for 24 tests.
End of tests n=8, 999 perm errors, 1000 next idx errors for 1000 tests.
End of tests n=10, 999 perm errors, 1000 next idx errors for 1000 tests.
xmsim: *W,RNQUIE: Simulation is complete.
```

In the unmodified assignment `perm` connects the permutation (`pdata_out`) output to the permutation input (`pdata_in`), which is wrong except for the identity permutation. That's why each module gets one permutation correct, as can be seen in the output above. (For example, for $n = 3$, 5 perm errors out of $3! = 6$ tests.)

The testbench always shows the first few outputs of each instance. For a correct assignment the output would include:

```
Starting tests for w=8, n=3
Trace of permutation: 0 0 0  -> a b c
Trace of permutation: 0 1 0  -> a c b
Trace of permutation: 1 0 0  -> b a c
Trace of permutation: 1 1 0  -> b c a
Trace of permutation: 2 0 0  -> c a b
Trace of permutation: 2 1 0  -> c b a
Finished with n=3, 0 perm errors, 0 next idx errors in 6 tests.
Starting tests for w=7, n=4
Trace of permutation: 0 0 0 0  -> a b c d
Trace of permutation: 0 0 1 0  -> a b d c
Trace of permutation: 0 1 0 0  -> a c b d
Trace of permutation: 0 1 1 0  -> a c d b
```

For $n = 3$ the testbench sets `pdata_in[0]='c'`, `pdata_in[1]='b'`, and `pdata_in[2]='a'`. The module needs to work for any settings for `pdata_in`, but the testbench sets `pdata_in` to values in `a,b,c,..` to make debugging easy.

Testbench output starting `Trace of permutation` shows the value of `pnum_in` (index, a factorial number) and `pdata_out` when `pdata_out` is correct. For the $n = 3$ module notice that all 6 permutations (permuted inputs) are shown, such as `a b c`. The sample above also shows the first few outputs of the $n = 4$ instance.

The digits of the permutation number are separated by spaces. The leftmost digit is (of course) the most significant, at position `n-1`. Being a permutation number, the least significant digit is always zero.

For each instance the first permutation is always identity (`pnum_in=0`), and the first 5 permutations are shown. For instances where $n! \leq 1000$ (or the value of `max_tests`) all permutations are tried. Otherwise, after showing 5 consecutive permutations a new random permutation is chosen. That can be seen below.

```
Starting tests for w=8, n=8
Trace of permutation: 0 0 0 0 0 0 0 0  -> a b c d e f g h
Trace of permutation: 0 0 0 0 0 0 1 0  -> a b c d e f h g
Trace of permutation: 0 0 0 0 0 1 0 0  -> a b c d e g f h
Trace of permutation: 0 0 0 0 0 1 1 0  -> a b c d e g h f
Trace of permutation: 0 0 0 0 0 2 0 0  -> a b c d e h f g
Trace of permutation: 5 2 3 2 1 0 0 0  -> f c e d b a g h
Trace of permutation: 5 2 3 2 1 0 1 0  -> f c e d b a h g
Trace of permutation: 5 2 3 2 1 1 0 0  -> f c e d b g a h
```

If there is an error the provided and correct outputs are shown. Here again is the output from the unmodified assignment:

```
Starting tests for w=8, n=3
Trace of permutation: 0 0 0  -> a b c
Error in next index:  0 0 0  -> 0 0 0   !=  0 1 0  (correct)
Error in permutation: 0 1 0  -> a b c   !=  a c b  (correct)
Error in next index:  0 1 0  -> 0 1 0   !=  1 0 0  (correct)
```

The first permutation output, `a b c`, is correct. The `pnum_out` value is wrong, that's shown in the line starting `Error in next index`. That line shows the value of `pnum_in` (the factorial number) to the left of the `->` and the value of `pdata_out` to the right of `->`. The correct value is shown to the right of `!=`. Similar information is shown for incorrect permutations.

Helpful Examples

An example that might help in computing `pnum_out` is from the class notes on generate statements. Module `ripple_w_r` recursively implements an adder. Pay attention to how `bfa` computes the LSB of the sum, and the recursive instance computes the remaining bits:

```
module ripple_w_r #( int w = 16 )
   ( output uwire [w-1:0] sum,   output uwire cout,
     input uwire [w-1:0] a, b,   input uwire cin);
   uwire c;

   // Instantiate a BFA to handle least-significant bit.
   //
   bfa bfa( sum[0], c, a[0], b[0], cin );

   if ( w == 1 )
     // If just one bit, we're done.
```

```
    //
    assign cout = c;
  else
    // Recursively instantiate this module to handle remaining bits.
    //
    ripple_w_r #(w-1) r(sum[w-1:1], cout, a[w-1:1], b[w-1:1], c);
endmodule
```

There's no need to use a BFA for this assignment. Use continuous assignments or procedural code to compute one digit of `pnum_out`.

In most examples of where we recursively describe a module we omit a particular bit (bit 0 in the ripple adder example above) in the connection to the recursive instance, or we have two recursive instances, each connected to half the inputs. The `perm` module is different because the digit to omit depends on the `pdata_in`. So, we need to use procedural code to compute an input to the recursive module and there are no good past assignment that do that. The closest is the Batcher merge module from 2018 Homework 5 where the odd and even elements of each of two the inputs were separated and recombined as inputs to two recursive instances:

```
module batcher_merge #( int n = 4, int w = 8 )
   ( output uwire [w-1:0] x[2*n], input uwire [w-1:0] a[n], b[n] );
   uwire [w-1:0] xlo[n], xhi[n];

   if ( n == 1 ) begin
      assign xlo[0] = a[0];
      assign xhi[0] = b[0];
   end else begin

      localparam int nh = n/2;
      uwire [w-1:0] ae[nh], ao[nh], be[nh], bo[nh];
      for ( genvar i=0; i<nh; i++ )
        begin
           assign ae[i] = a[2*i];
           assign ao[i] = a[2*i+1];
           assign be[i] = b[2*i];
           assign bo[i] = b[2*i+1];
        end

      batcher_merge #(nh,w) mlo( xlo, ae, bo );
      batcher_merge #(nh,w) mhi( xhi, ao, be );
   end

   for ( genvar i=0; i<n; i++ )
     sort2 #(w) s2( x[2*i], x[2*i+1], xlo[i], xhi[i] );
endmodule
```

Problem 1: Module `perm` has two data inputs, `pdata_in` and `pnum_in`. Input `pdata_in` is an array of `n` items, each `w` bits wide, where `n` and `w` are module parameters. Input `pnum_in` is an `n`-element array of `dw`-bit digits, where `dw` is a parameter. Module `perm` has three outputs, `pdata_out`, `pnum_out`, and `carry_out`. Like `pdata_in`, output `pdata_out` is an `n`-element array of `w`-bit items, and like `pnum_in`, output `pnum_out` is an `n`-element array of `dw`-bit digits. Output `carry_out` is one bit.

Output `pdata_out` is to be set to a permutation (rearrangement) of the elements of `pdata_in`. Suppose `pdata_in = {a,b,c}` (which means $n = 3$, and perhaps $w = 8$) and that the elements of `pdata_in` and `pdata_out` are ASCII characters. Then valid outputs could be `pdata_out = {a,b,c}`, `pdata_out = {b,a,c}`, etc. An invalid output would be `pdata_out = {a,a,c}`, it's invalid because `a` appears twice and `b` does not appear.

Input `pnum_in`, a factorial number, specifies how `pdata_in` should be permuted. A permutation is constructed iteratively, starting from the most-significant digit of `pnum_in`, which is `pnum_in[n-1]` and specifies where the value of `pdata_out[n-1]` should be drawn from. The Verilog code below (also part of the assignment file) shows how `pdata_out` is computed:

```
module perm_behavioral
  #( int w = 8, n = 20, dw = $clog2(n) )
   ( output logic [w-1:0] pdata_out[n], output logic [dw-1:0] pnum_out[n],
     output logic carry_out,
     input uwire [w-1:0] pdata_in[n], input uwire [dw-1:0] pnum_in[n] );

   always_comb begin

      pdata_out = pdata_in;

      for ( int i=n-1; i>0; i-- ) begin
         automatic logic [dw-1:0] pos = i-pnum_in[i];
         automatic logic [w-1:0] x = pdata_out[pos];
         for ( int j=pos; j<i; j++ ) pdata_out[j] = pdata_out[j+1];
         pdata_out[i] = x;
      end
   end
```

Notice that `pdata_out` is written multiple times, each iteration of the `i` loop permanently writes `pdata_out[i]` but also changes some other elements.

(*a*) Add code to `perm`, including recursive instantiation, so that `pdata_out` is a permutation of `pdata_in` as specified by `pnum_in`. Module `perm` with parameter `n>1` must recursively instantiate itself and the module must be synthesizable. Use command `genus -files syn.tcl` to synthesize.

(*b*) Add code to `perm`, including recursive instantiation, so that output `pnum_out` is the factorial number that follows `pnum_in`. The module must be synthesizable. As a reference the Verilog code below computes the next factorial number.

```
   always_comb begin
      // Compute next factorial (permutation) number.
      carry_out = 1;
      for ( int i=0; i<n; i++ ) begin
         automatic int radix = i + 1;
         automatic logic [dw:0] next_val = pnum_in[i] + carry_out;
         if ( next_val < radix ) begin
```

```
            pnum_out[i] = next_val;
            carry_out = 0;
         end else begin
            pnum_out[i] = 0;
         end
      end
   end
```

LSU EE 4755 Homework 4 Due: 6 November 2023

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, 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 to answer these questions. Just don't trust the answers. (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**.

Helpful Examples

See the simple model slides for material on computing cost and delay, and also for a list of some sample problems. Also see 2022 Homework 3.

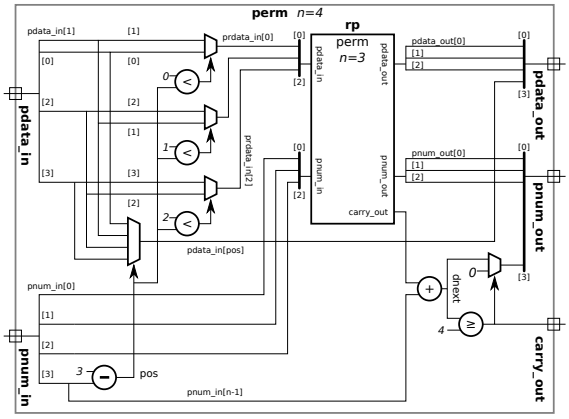
Permutation Module

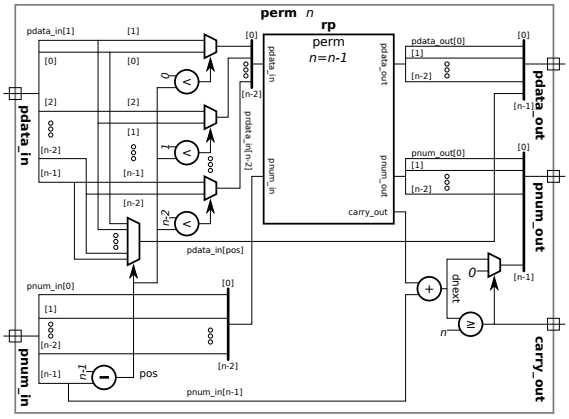
This assignment is based on the solution to Homework 3, the recursive permutation module `perm`, and the solution to Midterm Exam Problem 1, the inferred hardware for the permutation module. See Homework 3 for details on what the permutation module does. Appearing below is the Homework 3 solution with some comments removed. For the unabridged version visit `https://www.ece.lsu.edu/koppel/v/2023/hw03-sol.v.html`.

```
module perm
  #( int w = 8, n = 20, wd = $clog2(n) )
   ( output uwire [w-1:0] pdata_out[n],    output uwire [wd-1:0] pnum_out[n],
     output uwire carry_out,
     input uwire [w-1:0] pdata_in[n],       input uwire [wd-1:0] pnum_in[n] );

   if ( n == 1 ) begin

      assign pdata_out[0] = pdata_in[0];
      assign carry_out = 1;
      assign pnum_out[0] = 0;

   end else begin

      //  Set pos to the position of the element to be moved.
      uwire [wd-1:0] pos = n - 1 - pnum_in[n-1];

      //  Copy the element at position pos to position n-1 in the output.
      assign pdata_out[n-1] = pdata_in[pos];

      //  Prepare an array of n-1 elements and set to ..
      //  .. the elements of pdata_in except for the element at pos.
      uwire [w-1:0] prdata_in[n-1];
      for ( genvar i=0; i<n-1; i++ )
        assign prdata_in[i] =  i < pos  ?  pdata_in[i]  :  pdata_in[i+1];

      // Recursively instantiate perm.
      uwire co;
      perm #(w,n-1,wd) rp( pdata_out[0:n-2], pnum_out[0:n-2], co,
                           prdata_in, pnum_in[0:n-2] );

      //  Compute a tentative next value of digit n-1.
      uwire [wd-1:0] dnext = pnum_in[n-1] + co;

      //  Determine whether there is a carry.
      assign carry_out = dnext >= n;

      //  Set the next value of digit n-1 based on whether there is a carry.
      assign pnum_out[n-1] = carry_out ? 0 : dnext;

   end

endmodule
```

Permutation Module Inferred Hardware

Midterm Exam Problem 1 asked for the inferred hardware for the `perm` module instantiated with `n=4`. The solution appears below on the left. For this assignment the inferred hardware for a non-specific value of `n` will be needed, that is shown on the right.





There's no need to squint, the diagrams appear again in larger size at the end of this assignment. Also, SVG source for these modules are at `https://www.ece.lsu.edu/koppel/v/2023/mt-p1-sol.svg` and `https://www.ece.lsu.edu/koppel/v/2023/hw04-perm-gen.svg`.

Problem 1: Compute the cost and delay of the following arithmetic hardware from the `perm` module. Assume that ripple units are used for addition, subtraction, and comparison.

(*a*) Compute the cost and delay of the hardware computing `pos = n - 1 - pnum_in[n-1]` in terms of w_d, the value of parameter `wd`. Optimize for constants, including `n`.

☐ Cost of hardware in terms of w_d. ☐ Delay of hardware in terms of w_d.

☐ Optimize for constants, don't confuse elaboration-time computation with computation hardware.

(*b*) Compute the cost and delay of the hardware computing `dnext = pnum_in[n-1] + co` in terms of w_d, the value of parameter `wd`. Optimize for constants and for the size of `co`. Assume in this problem that `pnum_in` and `co` arrive at $t = 0$.

☐ Cost of hardware in terms of w_d. ☐ Delay of hardware in terms of w_d.

☐ Optimize considering the size of `co`. ☐ Optimize for constants, don't confuse elaboration-time computation with computation hardware.

(*c*) Compute the cost and delay of the hardware described by these lines:

```
uwire [wd-1:0] dnext = pnum_in[n-1] + co;
assign carry_out = dnext >= n;
```

Assume in this problem that `co` and `pnum_in` arrive at $t = 0$. The cost, of course, includes the cost of computing `dnext` in the previous part. The delay must be computed taking both lines into account.

☐ Cost of hardware in terms of w_d. ☐ Delay of `co` in terms of w_d.

☐ Optimize considering the size of `co`. ☐ Optimize for constants, don't confuse elaboration-time computation with computation hardware.

There are more problems on the next page.

Problem 2: In this problem consider the multiplexors with inputs connecting to `pdata_in`. (In the diagram they are the multiplexors on the upper-left including the 2-input muxes the n-input mux.) Call these the pdata multiplexors. In the solutions to the parts below use w for the value of parameter `w` and w_d for the value of parameter w_d.

(a) Compute the cost of the pdata multiplexors for a module instantiated at size $n = N$ including only the hardware in the `n=N` instantiation, not in the recursive instantiations. The answer should be in terms of N and w. *Hint: this is easy.*

☐ Cost of the pdata multiplexors at one level in terms of N, w, and (if needed) w_d.

(b) **This is important. Expect to expend brain energy. Don't skip.** Compute the total cost of the pdata multiplexors for an instantiation at size $n = N$ including the recursive instantiations all the way down. The answer should be in terms of N and w.

☐ Cost of the pdata multiplexors including recursive instantiations in terms of N, w, and (if needed) w_d.

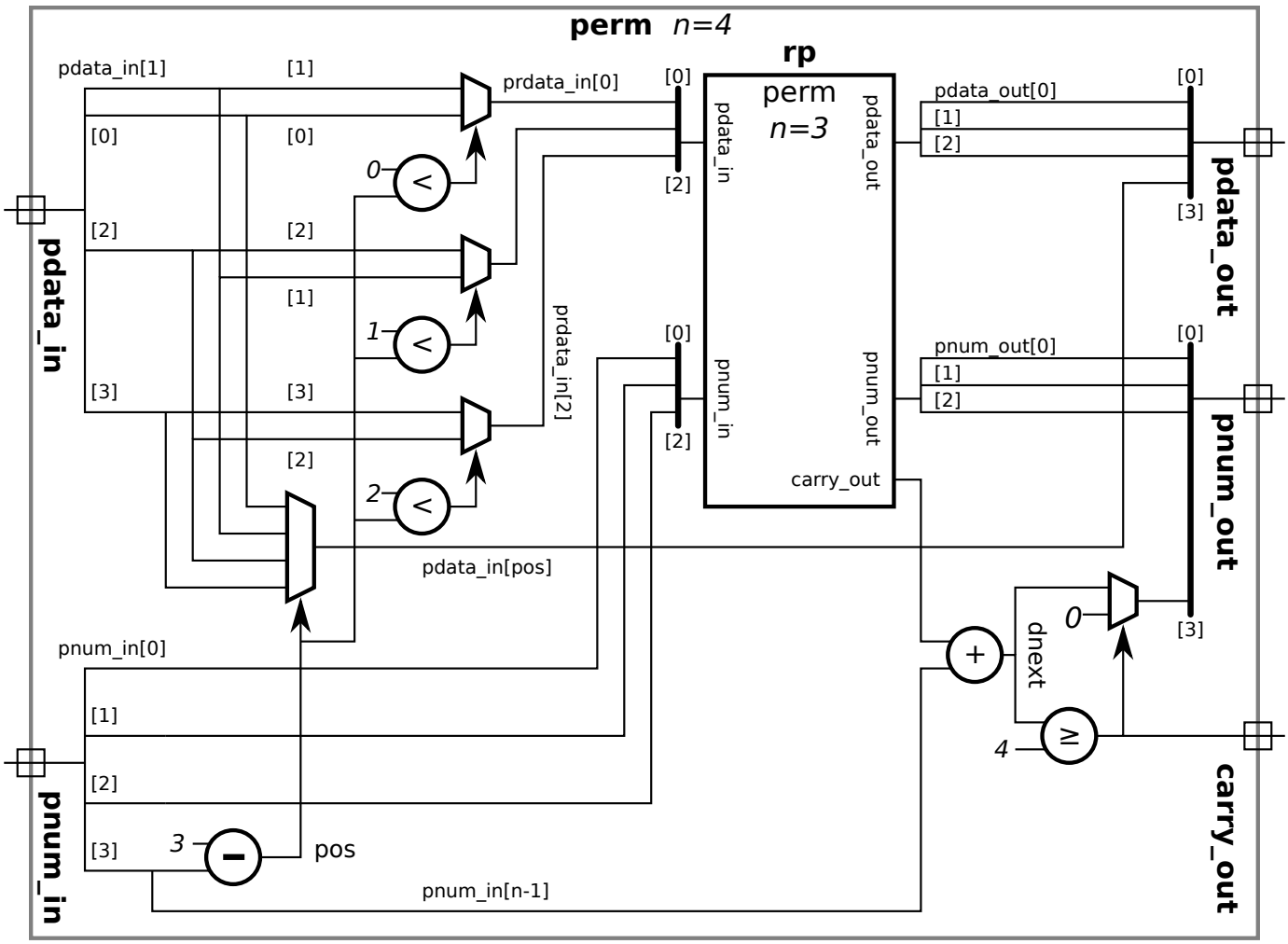
Problem 3: In this problem compute delays for `pdata_out` and `pnum_out`. In the solutions use d for the value of parameter `wd`. **This is also important and even more interesting. Expect to expend brain energy. Don't skip.**

(a) Assume that the delay of the subtractors computing `pos` is $\lg w_d$, where w_d is the value of parameter `wd`. (Note that $\lg w_d$ is not an answer to Problem 1.) Further, suppose the delay of the less-than units providing a select signal to the 2-input pdata multiplexors is zero. Using these assumptions compute the delay of the first and last elements of `pdata_out` for an instantiation at `n=N` and show the critical path. The delay should be in terms of N and w_d. To solve this problem it might be helpful to draw two instantiation levels to help find the critical path.

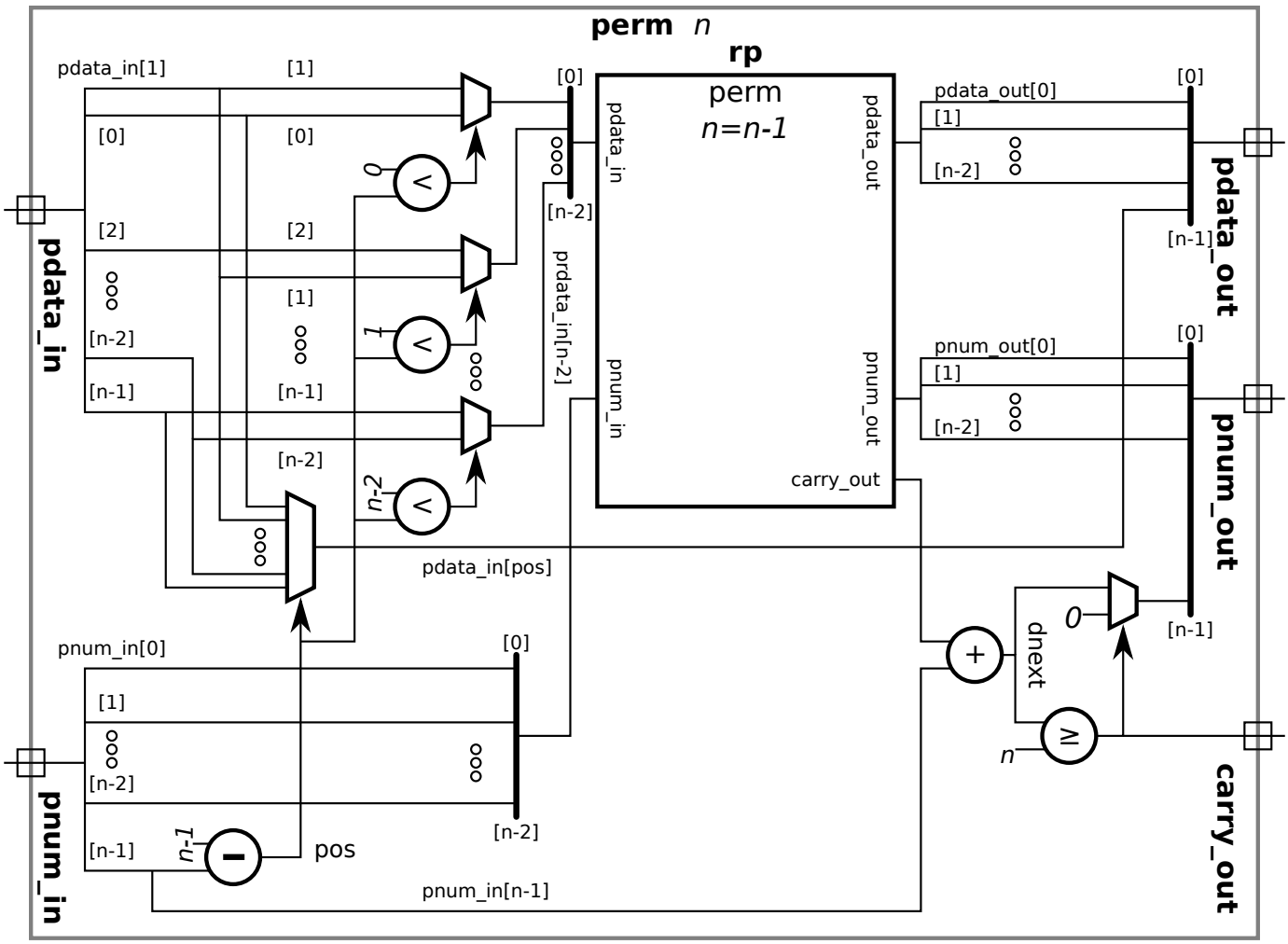
☐ Delay of `pdata_out[0]` in terms of N and w_d accounting for recursive instantiations. ☐ Show critical path.

☐ Delay of `pdata_out[N-1]` in terms of N and w_d accounting for recursive instantiations. ☐ Show critical path.

SVG source for the module below is at
`https://www.ece.lsu.edu/koppel/v/2023/mt-p1-sol.svg`.



SVG source for the module below is at
`https://www.ece.lsu.edu/koppel/v/2023/hw04-perm-gen.svg`.



LSU EE 4755 Homework 5 Due: 27 November 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/hw05.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, `hw03.v`.

Homework Overview

In previous assignments there were modules that permuted their inputs, the one called `pdata_in`. What would happen if the `pdata_in` input to our permutation module, `perm`, did not consist of n distinct elements? Let's suppose there would be some dire consequences that we need to avoid. That's what this assignment is about, module `uniq_vector_seq` will be used to determine if elements are distinct.

Module `uniq_vector_seq` has one `we`-bit data input, `element`, where `we` is a module parameter. There are two outputs, `n`-bit output `uniq_bvec` and `wc`-bit output `n_match`, where `n` and `wc` are module parameters. There is also a 1-bit input `start`.

At each positive clock edge a new element will be placed on input `element`. The module output `uniq_bvec` indicates the elements arriving in the prior n cycles (or since the last `start`) that appear only once. An element that appears only once is called *unique*. Output `uniq_bvec` (unique bit vector) has one bit for each of the past n cycles, with the least significant bit corresponding to the previous cycle. Let t denote the current cycle and let e_t denote the element at the `element` input in cycle t. The previous cycle is $t-1$, the one before that is $t-2$, and so on. (If this is starting to get confusing look at the examples in the description of the testbench.)

First, consider the case where `start=0` for at least the last n cycles. If `uniq_bvec[i]` is `1` then e_{t-i-1} is unique, meaning that $e_{t-i-1} \neq e_{t-j-1}$ for $i, j \in [0, n-1]$ and $i \neq j$. If `uniq_bvec[i]` is `0` then $e_{t-i-1} = e_{t-j-1}$ for some $i \neq j$.

For example, suppose $n = 4$ and suppose the most recent elements are 4, 7, 5, 5, with 4 the least recent of those. Then `uniq_bvec` will be 1100_2 because 5 appears twice. For 7, 7, 2, 2 `uniq_bvec` will be 0000_2, for 4, 7, 2, 0 `uniq_bvec` will be 1111_2, and finally for 3, 7, 7, 0 `uniq_bvec` will be 1001_2. The testbench shows the recent elements and the provided (module output) and if different, the correct value of `uniq_bvec`.

Output `n_match` should be set to the number of elements that the most recent element matches, including itself. For 4, 7, 5, 5 `n_match=2`, for 4, 7, 2, 0 `n_match=1` and for 9, 0, 9, 9 `n_match=3` but for 8, 8, 8, 6 `n_match=1`.

In a cycle where `start=1` the element on `element` starts a new sequence. So for the purposes of computing `uniq_bvec` and `n_match` `element` is considered not equal to any element that arrived in a previous cycle.

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 module `uniq_vector_seq`. The instantiations differ in `n` and in whether the `start` signal will be set to `1` during testing.

The testbench will always show information about at least 5 (or the value of `trace_len`) sets of inputs for each instantiation. If there are errors it will show information on at least 4 inputs that generate each kind of error.

Here is sample testbench output from a working module:

```
** Starting tests for n=4, input start used = No **
Trace, uniq_bvec: t=33,  1001
[  ,   ],  1,  0,  0,  1  <-- uniq_bvec
[13,  8],  7,  9,  9,  4  <-- Element
[ 0,  0],  0,  0,  0,  0  <-- Start
Trace, uniq_bvec: t=34,  0011
[  ,   ],  0,  0,  1,  1  <-- uniq_bvec
[ 8,  7],  9,  9,  4, 14  <-- Element
[ 0,  0],  0,  0,  0,  0  <-- Start
Trace, uniq_bvec: t=35,  1100
[  ,   ],  1,  1,  0,  0  <-- uniq_bvec
[ 7,  9],  9,  4, 14, 14  <-- Element
[ 0,  0],  0,  0,  0,  0  <-- Start
```

The text above shows information on three inputs, they occur at $t = 33$ through $t = 35$. (Actually those numbers refers to test numbers, not cycles.) The rows labeled `Elements` show the elements that have arrived over the past six cycles. The rightmost one is the most recent. The output above was for a module instantiated with `n=4`, so only the last 4 elements should matter. As an aid in debugging two additional elements are shown. So, for $t = 33$ the module should only pay attention to elements 7, 9, 9, 4, and the module should ignore 13, 8. The value of `uniq_bvec` is shown on the lines that start with `Trace`. The same value is shown in the rows labeled `uniq_bvec`. Note that the value `1001` is the output at $t = 33$. But the values shown in the `Element` and `Start` rows are from the past $n + 2$ cycles.

Notice that a bit of `uniq_bvec` is 0 if the corresponding element appears more than once. That is the case for 9 in the $t = 33$ input. At $t = 35$ the 9 element becomes uniq because the other 9 has arrived more than n cycles ago. For the examples above `n_matches` should be 1 at $t = 33$ and $t = 34$ and 2 at $t = 35$ (because there are two 14s).

The `start` input is used to reset the module. When `start=1` the prior elements are forgotten or ignored. The output below shows the correct effect of start.

```
** Starting tests for n=6, input start used = Yes **
Trace, uniq_bvec: t=53,  000111
[  ,   ],  0,  0,  0,  1,  1,  1  <-- uniq_bvec
[13, 99],  1,  1,  1, 19, 95, 53  <-- Element
[ 0,  0],  0,  0,  0,  0,  0,  0  <-- Start
Trace, uniq_bvec: t=54,  111111
[  ,   ],  1,  1,  1,  1,  1,  1  <-- uniq_bvec
[99,  1],  1,  1, 19, 95, 53, 19  <-- Element
[ 0,  0],  0,  0,  0,  0,  0,  1  <-- Start
```

```
Trace, uniq_bvec: t=55,  111111
[  ,   ],  1,  1,  1,  1,  1,  1  <-- uniq_bvec
[ 1,  1],  1, 19, 95, 53, 19, 32  <-- Element
[ 0,  0],  0,  0,  0,  0,  1,  0  <-- Start
```

At $t = 53$ in the output above there are three elements equal to 1, setting the `uniq_bvec` bits to zero. A $t = 54$ the `start` input is asserted and so the positions with 1 elements become unique. Also arriving elment 19 is also considered uniq. If in $t = 56$ a 19 arrived then the 19 elements would no longer be unique.

Here is testbench output for a module with errors:

```
** Starting tests for n=4, input start used = No **
Error, uniq_bvec: t=9,  0100!= 1100 ( correct )
[  ,   ], E0,  1,  0,  0  <-- uniq_bvec
[ 9,  9],  9,  2, 13, 13  <-- Element
[ 0,  0],  0,  0,  0,  0  <-- Start
Error, uniq_bvec: t=11,  0001!= 0011 ( correct )
[  ,   ],  0,  0, E0,  1  <-- uniq_bvec
[ 9,  2], 13, 13,  2,  1  <-- Element
[ 0,  0],  0,  0,  0,  0  <-- Start
```

The `Error` line shows first the value of `uniq_bvec` exiting the module, and then the correct value. The `uniq_bvec` line shows the value from the module, preceded with an `E` if that value is wrong. At $t = 9$ the MSB should have been a 1 because 9 is unique. Perhaps it is being dubbed not unique because there was another 9 earlier, but that should be too early to matter. A common mistake is to leave an output unconnected. The value would be shown as `x`, say `Ex` for a `uniq_bvec` bit.

Error lines are also shown if `n_match` is wrong:

```
** Starting tests for n=4, input start used = No **
Trace, uniq_bvec: t=10,  0000
Error:  n_match: 1 != 2 (correct)
[  ,   ],  0,  0,  0,  0  <-- uniq_bvec
[ 9,  9],  7, 13, 13,  7  <-- Element
[ 0,  0],  0,  0,  0,  0  <-- Start
```

In the output above `n_match` should have been 2 (since element 7 appears twice), but the module output is 1.

The testbench checks instantiations with two values of `n`, and does one set of tests where `start` is always 0 (after initialization) and another set of tests where `start` is occasionally set to 1.

At the end of the testbench a summary of error counts is printed:

```
End of tests n= 4, s=0: 0 bvec errors, 33368 n_match errors for 99992 tests.
End of tests n= 4, s=1: 0 bvec errors, 19771 n_match errors for 99992 tests.
End of tests n= 6, s=0: 0 bvec errors, 18240 n_match errors for 99988 tests.
End of tests n= 6, s=1: 0 bvec errors, 9326 n_match errors for 99988 tests.
xmsim: *W,RNQUIE: Simulation is complete.
```

The output above shows lots of `n_match` errors but no `bvec_uniq` errors.

Helpful Examples

The demo module computing a running sum will probably be most helpful. That and other pipelined modules are in file `pipe.v` in the homework file and can be viewed, with images, at `https://www.ece.lsu.edu/koppel/v/2023/pipe.v.html`. Look for module `simple_pipe_avg`.

Problem 1: In the unmodified assignment module `uniq_vector_seq` has some starter code, and it will actually generate the correct outputs for the `start=0` tests. It does so using combinational module `uniq_vector_comb`. The problem with the combinational module is that it is too costly, and also slow. Also, it ignores the `start` signal. So for this problem remove the instantiation of `uniq_vector_comb` from `uniq_vector_seq` and complete `uniq_vector_seq` so that it operates as described above. It is important that the cost is reasonable. The reason that `uniq_vector_comb` is costly is that it does n^2 comparisons. Module `uniq_vector_seq` should only perform about n comparisons per clock cycle.

(*a*) Add code to `uniq_vector_seq` so that `n_match` works as described above. The code must be synthesizable. Use command `genus -file syn.tcl` to synthesize. This part is easy.

- [ ] Complete module so that `n_match` is correct.
- [ ] Follow the checkbox items in `hw05.v`.

(*b*) Add code to `uniq_vector_seq` so that `uniq_bvec` works as described above. This is trickier, at least for a low-cost solution. It might be easier to get the `start=0` version working first.

- [ ] Complete module so that `uniq_bvec` is correct.
- [ ] Pay attention to cost, cost should not be proportional to n^2.
- [ ] Follow the checkbox items in `hw05.v`.

LSU EE 4755 Homework 6 Due: 1 December 2023

This assignment will be collected and graded, but the grades will not count.

Problem 1: Solve 2022 Final Exam Problem 1. In part a, a timing analysis is to be performed on a combinational vector normalization module `norm_comb`, and in part b a pipelined version of the module is to be designed.

Problem 2: Solve 2022 Final Exam Problem 2, in which Verilog code describing a vector normalization module, `norm_comb_n`, is to be completed.

Problem 3: Solve 2022 Final Exam Problem 3, in which a cost and timing analysis is to be done for an illustration of hardware for the `add_accum` module from 2019 Homework 6.

3 Fall 2022

LSU EE 4755 Homework 1 Due: 21 September 2022

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/2022/hw01.v.html`.

Problem 0: Following instructions at `https://www.ece.lsu.edu/koppel/v/proc.html`, set up your class account, copy the assignment, and run the Verilog simulator and synthesis program on the unmodified homework file, `hw01.v`. Do this early enough so that minor problems (*e.g.*, password doesn't work) are minor problems.

Homework Background

The goal of this homework assignment and follow-on assignments is to convert an ASCII string into a number. For example, to convert `"12"` (equivalently `'h3132`) into 12 (equivalently `'b1100` or `'d12` or `'hc`). An ASCII string is a sequence of bytes, but in this assignment there is just one byte. The follow-on assignments there will be multiple bytes.

The input to the module for this assignment, `atoi1`, is the character. The module has two outputs, the value, `val`, and whether the character is a valid digit. For example, `"1"` is a valid digit, but `"#"` is not.

The module has a parameter `r` which indicates the radix of the number that's expected. If `r=2` and the character is `"3"` then it is not a valid digit and the returned value should be zero. Further details are provided in the problem description below. For `r=16` the valid characters are 0 to 9, A to F, and a to f, with a and A, b and B, ... treated equivalently. The module should work for any `r` up to 36. As of this writing the testbench evaluates radices 4, 8, 10, 14, 16, 19. The TA-bot might test with different radices. Feel free to modify the testbench to try different radices. (Search for `testbench` and figure out the code.)

This assignment exercises basic Verilog skills like instantiating modules and understanding the difference between structural and procedural code. In the follow-on assignment the `atoi` modules will be connected to handle longer strings.

Testbench

To run compile your code and run the testbench press `F9` in an Emacs buffer in a properly set up account. In an unmodified assignment the testbench will generate output that includes the following near the end:

```
Radix  4, done with 256 tests, 0 val errors, 0 is_digit errors.
Radix  8, done with 256 tests, 0 val errors, 0 is_digit errors.
Radix 10, done with 256 tests, 0 val errors, 0 is_digit errors.
Radix 14, done with 256 tests, 0 val errors, 0 is_digit errors.
Radix 16, done with 256 tests, 0 val errors, 0 is_digit errors.
Radix 19, done with 256 tests, 0 val errors, 0 is_digit errors.
xmsim: *W,RNQUIE: Simulation is complete.
xcelium> exit
Total number of errors: 0
```

There are zero errors because the procedural code in `atoi1` is correct. Notice that there are separate tallies for each radix plus a grand total. Detailed messages are printed for the first few errors, after which only a tally is provided. For example, here is what the error messages would look like if the conversion to upper case were wrong:

```
xcelium> run
Radix  4, done with 256 tests, 0 val errors, 0 is_digit errors.
```

```
Radix  8, done with 256 tests, 0 val errors, 0 is_digit errors.
Radix 10, done with 256 tests, 0 val errors, 0 is_digit errors.
R 14  Error val 'h0d or D != A (correct) for string " a"
R 14  Error val 'h00 or 0 != B (correct) for string " b"
R 14  Error is_digit 0 != 1 (correct) for string " b"
R 14  Error val 'h00 or 0 != C (correct) for string " c"
R 14  Error is_digit 0 != 1 (correct) for string " c"
R 14  Error val 'h00 or 0 != D (correct) for string " d"
R 14  Error is_digit 0 != 1 (correct) for string " d"
Radix 14, done with 256 tests, 4 val errors, 3 is_digit errors.
R 16  Error val 'h0d or D != A (correct) for string " a"
R 16  Error val 'h0e or E != B (correct) for string " b"
R 16  Error val 'h0f or F != C (correct) for string " c"
R 16  Error val 'h00 or 0 != D (correct) for string " d"
R 16  Error is_digit 0 != 1 (correct) for string " d"
R 16  Error is_digit 0 != 1 (correct) for string " e"
R 16  Error is_digit 0 != 1 (correct) for string " f"
Radix 16, done with 256 tests, 6 val errors, 3 is_digit errors.
R 19  Error val 'h00d or D != A (correct) for string " a"
R 19  Error val 'h00e or E != B (correct) for string " b"
R 19  Error val 'h00f or F != C (correct) for string " c"
R 19  Error val 'h010 or G != D (correct) for string " d"
R 19  Error is_digit 0 != 1 (correct) for string " g"
R 19  Error is_digit 0 != 1 (correct) for string " h"
R 19  Error is_digit 0 != 1 (correct) for string " i"
Radix 19, done with 256 tests, 9 val errors, 3 is_digit errors.
xmsim: *W,RNQUIE: Simulation is complete.
xcelium> exit
Total number of errors: 28
```

Consider one of those lines with some ASCII-art underlining:

```
R 14  Error val 'h0d or D != A (correct) for string " a"
0000            1111    2    3                       44  <- ASCII art underlining
```

The part underlined with `0000` indicates that this result is for radix `r=14`. The character to be converted is a, though it's called a string and shown with a leading space. That's the part underlined with `44`. The part underlined with `3` is what the value should be in radix 14, and the part underlined with `2` is what the `atoi1` module's `val` output is, in radix 14. The part underlined in `1111` is the module output in hexadecimal. In this case, the value should be 10 (decimal) or A (base 14), but the module output is D (which is 13 in decimal).

That's one error line. Going back to the more complete testbench output notice that only strings with lower-case letters are wrong. This is what one would expect since we intentionally broke the lower-to-upper conversion.

The testbench only shows details for the first 4 errors of each type at each radix. If you want to see more errors feel free to edit the testbench. Search for `err < 5`. Feel free to edit the testbench in other ways to facilitate debugging. The TA-bot will run your code using its own testbench, so don't worry about being accused of cheating by modifying the testbench.

Problem 1: Appearing below (and in the assignment file `hw01.v`) is module `atoi1`, *ASCII to Integer of 1 character*. The module has an 8-bit input `char`, a 1-bit output `is_digit`, and a w-bit output `val`. There are also two parameters, `w` (width) and `r` (radix). Output `is_digit` is set to `1` iff (if and only if) `str` is a radix-r digit. If `char` is a digit output `val` is set to its value, otherwise `val` is set to zero.

```
module atoi1 #( int r = 32, w = $clog2(r) )
   ( output logic [w-1:0] val,  output uwire is_digit,  input uwire [7:0] char );
   logic [7:0] char_uc;
   logic [w-1:0] val_09, val_az;
   logic is_09, is_az;

   digit_valid_09 #(r,w) v09( is_09, val_09, char );
   assign is_digit = is_09 || is_az;

   always_comb begin
      char_uc = char >= Char_a && char <= Char_z ? char - Char_a + Char_A : char;
      val_az = 10 + char_uc - Char_A;
      is_az =  char_uc >= Char_A  &&  char_uc < Char_A + r - 10;
      if ( is_09 )       val = val_09;
      else if ( is_az )  val = val_az;
      else               val = 0;
   end
endmodule
```

For example, suppose `w=4` and `r=10`. If `char=51` (ASCII for the digit 3), then output `val` is set to 3 and `is_digit` is set to 1. If `char=58` (ASCII for `:` [colon]), then output `val` is set to 0 and `is_digit` is set to 0. If `char=65` (ASCII for `A`), then output `val` is set to 0 and `is_digit` is set to 0. Now suppose that `atoi1` is instantiated with `r=16` (hexadecimal). If `char=65` (ASCII for `A`), then output `val` is set to 10 and `is_digit` is set to 1. If `char=97` (ASCII for `a`), then output `val` is also set to 10 and `is_digit` is set to 1.

Module `atoi1` includes an instantiation of module `digit_valid_09`, a continuous assignment (of `is_digit`), and procedural code. Module `digit_valid_09`, which is finished, converts an ASCII character into a value if the character is a digit from 0 to 9, and if the value is valid (less than `r`). (Those who are not sure what `digit_valid_09` is doing might want to inspect module `atoi1_behavioral`, which uses only procedural code.)

Make the following changes to `atoi1`: Instantiate module `char_to_uc` (character to upper case) and use it to convert `char` to upper case. Instantiate module `digit_valid_az` and use it to compute `is_az` and `val_az`. Instantiate `mux2` modules and use them to route the correct value to the `val` output of `atoi1`. As you instantiate and connect these modules remove the procedural code that's no longer needed.

Also, add code to `digit_valid_az` and `char_to_uc` so that they compute their proper values.

To help with debugging, do this in small steps. For example, first complete the `char_to_uc` module and make sure there are no compilation errors. Then instantiate it in `atoi1`, and make sure there are no compilation errors and no testbench errors.

Pay attention to compilation errors and ask for help with any that you can't understand.

The code must by synthesizable. To synthesize your code issue the command `genus -files syn.tcl`. If there are no errors, running this command will generate output that includes like the following:

Module Name	Area	Delay Actual	Delay Target
atoi1_r2	1796	0.454	10.000 ns
atoi1_behavioral_r2	1796	0.454	10.000 ns
atoi1_r8	2047	0.495	10.000 ns
atoi1_behavioral_r8	2047	0.495	10.000 ns
atoi1_r10	2517	0.529	10.000 ns
atoi1_behavioral_r10	2517	0.529	10.000 ns
atoi1_r16	5792	0.752	10.000 ns
atoi1_behavioral_r16	5792	0.752	10.000 ns
atoi1_r2_3	3754	0.274	0.100 ns
atoi1_behavioral_r2_4	3754	0.274	0.100 ns
atoi1_r8_3	5762	0.260	0.100 ns
atoi1_behavioral_r8_4	5762	0.260	0.100 ns
atoi1_r10_3	7371	0.259	0.100 ns
atoi1_behavioral_r10_4	6937	0.260	0.100 ns
atoi1_r16_3	18302	0.363	0.100 ns
atoi1_behavioral_r16_4	18302	0.363	0.100 ns

The synthesis script is synthesizing both the module for this assignment, `atoi1`, and the behavioral version, `atoi1_behavioral`. The radix at which it is instantiated is appended to the name.

LSU EE 4755 Homework 2 Due: 7 October 2022

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/2022/hw02.v.html`.

Problem 0: Following instructions at `https://www.ece.lsu.edu/koppel/v/proc.html`, set up your class account, copy the assignment, and run the Verilog simulator and synthesis program 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 Background

This assignment is a follow-on to Homework 1, in which the `atoi1` modules will be used to convert an ASCII string holding a number into a value. For example, to convert `"12"` (equivalently `'h3132`) into 12 (equivalently `'b1100` or `'d12` or `'hc`). An ASCII string is a sequence of bytes, in this assignment there can be one or more bytes.

The string is on module input `str` and it is declared so that `str[0]` is the rightmost (least-significant) character of the ASCII string. For example, if the string were `" 987"` then `str[0]` would be the 7 (ASCII value $48+7=55$), `str[1]` would be the 8, `str[2]` the 9, and `str[3]` the space (ASCII value 32).

Let n denote the number of characters in the string. The ASCII number may take up n or fewer characters. For example, for $n=4$ the number 1 would only need one character. The remaining characters can be any non-digit character. For example for the 1 in radix 10 the string can be `" 1"`, or `"abc1"`, but not `"ab21"` since that would be the number 21.

The input to the modules for this assignment, `atoi_it` and `atoi_tr`, is the string. The modules have two outputs, the value, `val`, and the number of digits in the number, `nd`. For example, for input `"9 43"` in radix 10 the value is 43 and the number of digits is 2. The `9` does not count because it is separated by a non-digit character from the `43`. For radix 16 and input `" a12"` the value is 2578 and the number of digits is 3. If the radix is 3 and the string is `"32"` then the value is 2 and the number of digits is 1. The 3 is not a valid digit in trianary, and so it ends the number.

For `r=16` the valid characters are 0 to 9, A to F, and a to f, with a and A, b and B, ... treated equivalently. The module should work for any `r` up to 36.

As of this writing the testbench evaluates radices 10 and 16 and a variety of string lengths. Feel free to modify the testbench to try different radices. (Search for `testbench` and figure out the code.)

Reference Module

To help you get started, there is a reference module, `atoi_pr`, that correctly computes the value of a string. This module would not be a correct solution to either problem.

```
module atoi_pr
  #( int r = 11, n = 5, wv = $clog2( r**n ), wd = $clog2(n+1) )
   ( output logic [wv-1:0] val,
     output logic [wd-1:0] nd,
     input uwire [7:0] str [n-1:0] );

   always_comb begin
      val = 0; nd = 0;
      for ( int i=0; i<n; i++ ) begin
         // Get val of current char. If val is < 0 then char is not a digit.
         automatic int dval = atoi1_func(str[i],r);
```

```
            if ( dval < 0 ) break;
            val += dval * r**i;
            nd++;
        end
    end
endmodule
```

Testbench

To compile your code and run the testbench press F9 in an Emacs buffer in a properly set up account. In an unmodified assignment the testbench will generate output that includes the following near the end:

```
Total errors for radix 10: 14000 len, 14140 val
Total errors for radix 16: 14000 len, 14224 val
Total errors for string length  1:  4000 len,  4052 val
Total errors for string length  2:  4000 len,  4052 val
Total errors for string length  3:  4000 len,  4052 val
Total errors for string length  4:  4000 len,  4052 val
Total errors for string length  7:  4000 len,  4052 val
Total errors for string length  8:  4000 len,  4052 val
Total errors for string length  9:  4000 len,  4052 val
Total errors for mod atoi_it: 14000 len, 14182 val
Total errors for mod atoi_tr: 14000 len, 14182 val
```

The errors are tallied above three ways: by radix, by string length, and by module (`atoi_it` and `atoi_tr`). In the output above both modules have errors, and their are errors at each radix and length. In the output below module `atoi_it` has zero errors, and errors only occur at lengths 3, 7, 9. The errors would have to be due to `atoi_tr`:

```
Total errors for radix 10:  1201 len,  1201 val
Total errors for radix 16:  1144 len,  1036 val
Total errors for string length  1:     0 len,     0 val
Total errors for string length  2:     0 len,     0 val
Total errors for string length  3:   687 len,   687 val
Total errors for string length  4:     0 len,     0 val
Total errors for string length  7:  1434 len,  1434 val
Total errors for string length  8:     0 len,     0 val
Total errors for string length  9:   224 len,   116 val
Total errors for mod atoi_it:     0 len,     0 val
Total errors for mod atoi_tr:  2345 len,  2237 val
Total number of errors: 4582
```

The messages above are tallies printed near the end. Detailed messages are printed for the first few errors. Here are two error messages (of many from the same run as above:

```
Mod-atoi_tr R-10 n- 7 Ty-SP  Error val 1 != 2011 (correct) for string "   2011"
Mod-atoi_tr R-10 n- 7 Ty-SP  Error len 1 != 4 (correct) for string "   2011"
```

Each of the two lines indicates that the error was with module `atoi_tr` instantiated at `r=10` (radix 10) and string length of $n = 7$. (Don't confuse string length with the length of the number in the string.) `Ty-SP` indicates the type of test, in this case a number padded with spaces. The first line indicates that the value should have been 2011 but the module output was 1. The second line informs us that the length should have been 4, but the module `nd` output was 1.

There are three types of tests: `Ty-SC`, `Ty-SP`, and `Ty-GE`. For `Ty-SC` tests the number is always

one digit (regardless of the string length). For `Ty-SP` tests the number is followed spaces. For `Ty-GE` the number is followed by any non-digit character.

The testbench only shows details for the first 4 errors of each type at each radix. If you want to see more errors feel free to edit the testbench. Search for `err < 5`. Feel free to edit the testbench in other ways to facilitate debugging. The TA-bot will run your code using its own testbench, so don't worry about being accused of cheating by modifying the testbench.

Similar Problems

See the l025-gen-elab.v demo code for examples of how to use generate statements iteratively (needed for Problem 1) and recursively (needed for Problem 2). An easy example is `ripple_w` from that set. Pay attention to how the carry signals are connected from one BFA to the other:

```
module ripple_w
  #( int w = 4 )
   ( output uwire [w-1:0] sum,   output uwire cout,
     input uwire [w-1:0] a, b,   input uwire cin);

   uwire         c[w-1:-1];
   assign        c[-1] = cin;
   assign        cout = c[w-1];

   for ( genvar i = 0; i<w; i++ )
     bfa bfai( sum[i], c[i], a[i], b[i], c[i-1] );
endmodule
```

A simple recursive module is `min_t` which finds the minimum of its `n` inputs:

```
module min_t
  #( int w = 4,  n = 8 )
   ( output uwire [w-1:0] e_min,   input uwire [w-1:0] e [ n-1:0 ] );

   if ( n == 1 ) begin

      assign e_min = e[0];

   end else begin

      localparam int n_lo = n / 2;
      localparam int n_hi = n - n_lo;

      uwire [w-1:0] m_lo, m_hi;

      min_t #(w,n_lo) mlo( m_lo, e[n_lo-1:0] );
      min_t #(w,n_hi) mhi( m_hi, e[n-1:n_lo] );

      min_2 #(w) m2( e_min, m_lo, m_hi);
   end
endmodule
```

See the count-leading-zeros assignment from 2019 Homework 2 for an example of how to recursively instantiate a module and combine results.

Problem 1: Module `atoi_it` has an `n`-character input `str`, and outputs `val` (value) and `nd` (number of digits), as well as parameters `r` (radix) and `n` (number of characters in string). Following the rules further below, complete module `atoi_it` so that `val` is the value of the radix-`r` ASCII representation of a number in `str` and `nd` is set to the number of digits in the number (not to be confused with the number of characters in the string). Further details are described in the background section above.

Module `atoi_it` must use instantiations of module `atoi1` to convert characters to their values and it must use instantiations of `mult_by_c` to do multiplication by a constant. The module may also instantiate `add` and `mux2` modules, but it doesn't have to. A selection of modules is defined under the Problem 0 section of `hw02.v`.

Module `atoi_it` must not instantiate itself (that's Problem 2). Instead, use a generate loop to instantiate the `atoi1` and `mult_by_c` modules.

To help you get started, module `atoi_it` includes an instantiation of `atoi1` and `mult_by_c`. But, those are not in a generate loop and won't work. They are only there to show you how to instantiate something correctly.

Make sure that your module is synthesizable by running the synthesis script. The command is `genus -files syn.tcl`.

Problem 2: Module `atoi_tr` has the same ports and parameters as `atoi_it` and should produce the same outputs. Complete `atoi_tr` so that it does so by recursively instantiating two instances of itself, with each instance operating on about half of the string. As with `atoi_it`, it must use instantiations of `atoi1` to convert characters and `mult_by_c` to perform multiplication. Make sure that the module is synthesizable.

Some may have realized (or will come to realize) that for certain radices neither multiplication nor addition (at least for values) is needed. Don't worry about that, it's okay to use `mult_by_c` even when not needed.

The module must be synthesizable. See the comments in the code for other requirements and things to look out for.

LSU EE 4755 Homework 3 Due: 17 Oct 2022, 11:30 CDT

Resources

To help with this assignment review the simple cost model slides and the material in generate statement demo code.

The following problems ask for both inferred hardware and a cost/performance analysis: 2019 Midterm Exam Problem 3c (equality module with shifted inputs), 2021 Midterm Exam Problem 2 (a concentrator for neural network hardware reading sparse weights).

The following are good cost and performance analysis questions (these are the same ones mentioned in the simple model slides): The "find oldest" (big mux) problem covered in class can be found in 2017 Final Exam Problem 3, the knapsack problem hardware covered in class can be found in 2016 Final Exam Problem 2 and 4.

The following are good inferred hardware and optimization problems. Start with 2019 Midterm Exam Problem 1 (a recursively described clz [count leading zeros] module). A problem combining both recursive and iterative generate statements can be found in 202 Midterm Exam Problem 4.

A sequential version of the ASCII-to-value hardware was also assigned in this course. The hardware was described by procedural code and it operated sequentially, so I don't suggest that it specifically be studied for clues on how to solve this assignment.

Problem 1: Compute the cost and delay, using the simple model, of the `atoi1` module (from the solution to Homework 1) instantiated with `r=12`. Base this on a module with reasonable optimizations applied and be sure to account for constants when computing cost and delay.

- Base your analysis of ripple implementations of the adder and magnitude comparison units.
- Show cost.
- Show delay of each output and identify the critical path.
- **Account for constants** when computing cost and delay.

```
module atoi1
  #( int r = 32, w = $clog2(r) )
   ( output logic [w-1:0] val,    output logic is_digit,
     input uwire [7:0] char );

   logic [w-1:0] val_09, val_az, val_n;
   logic is_09, is_az;

   digit_valid_09 #(r,w) v09( is_09, val_09, char );
   uwire [7:0] char_uc;
   char_to_uc tuc(char_uc,char);
   digit_valid_az #(r,w) vaz( is_az, val_az, char_uc );

   uwire [w-1:0] z = 0;
   mux2 #(w) mval(val_n,is_09,val_az,val_09);
   mux2 #(w) mval0(val,is_digit,z,val_n);
```

```
   assign is_digit = is_09 || is_az;
endmodule

typedef enum
  { Char_0 = 48, Char_9 = 57, Char_A = 65, Char_Z = 90, Char_a = 97, Char_z = 122 }
  Chars_Special;

module digit_valid_09
  #( int r = 9, vw = $clog2(r) )
   ( output uwire valid, output uwire [vw-1:0] val, input uwire [7:0] char );
   assign val = char - Char_0;
   assign valid = char >= Char_0 && char <= Char_9 && char < Char_0 + r;
endmodule

module char_to_uc( output uwire [7:0] uc, input uwire [7:0] c );
   uwire is_lc = c >= Char_a && c <= Char_z;
   uwire [7:0] uc_if_lc = c - Char_a + Char_A;
   mux2 #(8) m( uc, is_lc, c, uc_if_lc );
endmodule

module digit_valid_az
  #( int r = 11, vw = $clog2(r) )
   ( output uwire valid, output uwire [vw-1:0] val, input uwire [7:0] char );
   assign val = 10 + char - Char_A;
   assign valid = char >= Char_A && char < Char_A + r - 10;
endmodule

module mux2
  #( int w = 3 )
   ( output uwire [w-1:0] x,
     input uwire s,   input uwire [w-1:0] a0, a1 );
   assign x = s ? a1 : a0;
endmodule
```

Problem 2: Appearing further below is the `atoi_it` from the solution to Homework 2.

(*a*) Show the hardware inferred for an `atoi_it` module instantiated with `r=14` (yes, radix 14) and `n=3`.

- Show `atoi1`, `mult_by_c`, and `add` instances as modules, do not show what is inside.
- Show the hardware inferred for the operators, such as `&&` and `?:`.
- Do not confuse parameters and ports.
- Omit hardware that does not belong, such as "hardware" to compute values needed at elaboration time.
- Be sure to show the inferred logic. Remember that generate statements describe what happens at elaboration time, not what happens at simulation time nor does it describe operations performed by the hardware.

(*b*) Show the hardware inferred for an `atoi_it` module instantiated with `r=16` (hexadecimal this time) and `n=3`, and show the hardware after optimization. Consider the impact of optimization on the `mult_by_c` and `add` modules, which should be considerable since `r` is a power of 2.

```
module atoi_it
  #( int r = 11, n = 5, wv = $clog2( r**n ), wd = $clog2(n+1) )
   ( output logic [wv-1:0] val,
     output logic [wd-1:0] nd,
     input uwire [7:0] str [n-1:0] );

   uwire [wv-1:0] vali[n-1:-1];
   uwire is_valid[n-1:-1];
   uwire [wd-1:0] ndi[n-1:-1];
   assign is_valid[-1] = 1;
   assign ndi[-1] = 0;
   assign vali[-1] = 0;
   assign nd = ndi[n-1];
   assign val = vali[n-1];

   localparam int wcv = $clog2(r);

   for ( genvar i=0; i<n; i++ ) begin

      // Find Value of Digit i
      //
      uwire [wcv-1:0] valdr;
      uwire is_digit;
      atoi1 #(r,wcv) a( valdr, is_digit, str[i] );

      // Determine if this digit continues a sequence of valid digits
      // starting at str[0].
      //
      assign is_valid[i] = is_digit && is_valid[i-1];

      // Replace value with zero if str[i] is not a digit, or if the
      // string of valid digits has already ended.
      //
      uwire [wcv-1:0] vald = is_valid[i] ? valdr : 0;

      // Multiply (scale) the digit value based on its position in the number.
      //
      uwire [wv-1:0] vals;
      mult_by_c #( .w_in(wcv), .c(r**i), .w_out(wv) ) mc( vals, vald );

      // Add the scaled digit to the value accumulated so far.
      //
      add #(wv) a1( vali[i], vali[i-1], vals );

      // Update the number of digits so far.
      //
      assign ndi[i] = is_valid[i] ? i+1 : ndi[i-1];
   end
endmodule
```

Problem 3: Appearing further below is the `atoi_tr` from the solution to Homework 2. Show the inferred logic for an instantiation with `r=10` and `n=9`.

- Show the logic for one level. That is, show the two instantiations of `atoi_tr`, `alo` and `ahi`, but don't show what is inside of `alo` nor `ahi`.
- Show the `mult_by_c` instantiations as modules, do not show what is inside.
- Show the hardware inferred for the operators, such as `&&` and `?:`.
- Omit hardware that does not belong, such as "hardware" to compute values needed at elaboration time.
- Do not confuse parameters and ports.
- Be sure to show the inferred logic. Remember that generate statements describe what happens at elaboration time, not what happens at simulation time nor does it describe activities performed by the hardware.

```
module atoi_tr
  #( int r = 11, n = 5, wv = $clog2( r**n ), wd = $clog2(n+1) )
   ( output uwire [wv-1:0] val,   output var logic [wd-1:0] nd,
     input uwire [7:0] str [n-1:0] );

   if ( n == 1 ) begin

      uwire is_dd;
      uwire [wv-1:0] valr;
      atoi1 #(r,wv) a( valr, is_dd, str[0] );
      assign val = is_dd ? valr : 0;
      assign nd = is_dd; // Note: nd may be more than one bit.

   end else begin

      // Prepare to split the input string into two halves. Note that
      // the hi half may be larger, and so we use nhi to compute the
      // number of bits needed in the value output (vwh) and the
      // number of digits output (dwh).
      //
      localparam int nlo = n/2;
      localparam int nhi = n - nlo;
      localparam int vwh = $clog2( r**nhi );
      localparam int dwh = $clog2( nhi+1 );
      //
      uwire [vwh-1:0] vallo, valhi;
      uwire [dwh-1:0] ndlo, ndhi;

      // Split input string between two recursive instantiations
      //
      atoi_tr #(r,nlo,vwh,dwh) alo( vallo, ndlo, str[nlo-1:0] );
      atoi_tr #(r,nhi,vwh,dwh) ahi( valhi, ndhi, str[n-1:nlo] );

      // Determine whether the hi half of the string may be part
      // of the number.
      //
      uwire hitoo = ndlo == nlo;
      uwire [vwh-1:0] valhid = hitoo ? valhi : 0;

      // Scale the upper half.
      //
      uwire [wv-1:0] valhis;  // Value High Scaled
      mult_by_c #(vwh,r**nlo,wv) mc( valhis, valhid );

      assign val = vallo + valhis;
      assign nd = hitoo ? nlo + ndhi : ndlo;
   end
endmodule
```

LSU EE 4755 Homework 4 Due: 4 November 2022

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/2022/hw04.v.html`.

Problem 0: Following instructions at `https://www.ece.lsu.edu/koppel/v/proc.html`, set up your class account, copy the assignment, and run the Verilog simulator and synthesis program on the unmodified homework file, `hw04.v`. Do this early enough so that minor problems (*e.g.*, password doesn't work) are minor problems.

Helpful Past Homework Assignments

For those who would like to see a fairly simple sequential circuit, and one that counts characters, see 2017 Homework 4, maxrun.

Problem 1: Module `word_count` has three inputs, an 8-bit `char` input, and 1-bit inputs `clk` and `reset`. At each positive edge of `clk` a new ASCII character will be available at input `char`. The characters might be from a text file, a keyboard, or some other source of English text. Based on the word rules given below these characters form words, and the module is to count the words and provide other information.

Module `word_count` has three parameters, `wl`, `wn`, and `n_avg_of`. The module has six outputs. Output `len_word`, which is `wl` bits, is the length so far of the current word, or the length of the most recent word. Output `n_words`, which is `wn` bits, is the number of complete words counted since the last reset.

Output `len_avg`, which is also `wl` bits, is the average length of the `n_avg_of` most recent completed words with the fractional part truncated. **If fewer than** `n_avg_of` words have ended since the last reset then `len_avg` should be zero. For example, if `n_avg_of=4` and the lengths of the four most recent words are 8, 4, 12, and 15 then `len_avg` should be set to $\lfloor(8+4+12+15)/4\rfloor = \lfloor 39/4\rfloor = \lfloor 9.75\rfloor = 9$. If there is a reset and then only three words have ended, `len_avg` should be `0`.

Output `word_start` should be set to `1` iff the current character starts a word. Output `word_part` should be set to `1` if the current character is part of a word based on the word rules described further below. (If `word_start` is `1` then `word_part` is `1`.) Output `word_ended` is 1 if the character in the previous cycle was the last character of a word.

For an example of how these output should be set examine the testbench output below, collected for the text "A or bee":

```
       W-M    I  Text---->!          SPE L  N A {D}
Trace  2-5    0 "          A"    ___ SP_ 1  0 0 {1}
Trace  2-5    1 "         A "    sp_ __E 1  1 0 {0}
Trace  2-5    2 "        A o"    __e SP_ 1  1 0 {1}
Trace  2-5    3 "       A or"    sp_ _P_ 2  1 0 {1}
Trace  2-5    4 "      A or "    _p_ __E 2  2 1 {0}
Trace  2-5    5 "     A or b"    __e SP_ 1  2 1 {1}
Trace  2-5    6 "    A or be"    sp_ _P_ 2  2 1 {1}
Trace  2-5    7 "   A or bee"    _p_ _P_ 3  2 1 {1}
Trace  2-5    8 " A or bee "    _p_ __E 3  3 2 {0}
Trace  2-5    9 "A or bee 2"    __e ___ 3  3 2 {0}
Trace  2-5   10 " or bee 2n"    ___ ___ 3  3 2 {1}
```

Each line shows the output at one cycle, the `I` column shows an index (which is something like a cycle number). The `W` column shows the value of `n_avg_of` and the `M` column shows the maximum possible word length. The last column, `{D}`, is for debugging, see the discussion further below.

The most-recent ten characters are shown under the `Text` heading, in the first line (index 0), `A` is the most recent character. There will be an `R` to the right of the text in a cycle when `reset` is `1`.

The `L` column shows the length of the word so far, or the length of the most recent word. The `N` column shows the number of words (incremented when the word ends), and the `A` column shows a running average of the last word lengths, the last 2, in this case. The column headed `SPE` shows the state of the outputs of `word_start`, `word_part`, and `word_ended` outputs. An upper case letter shows the state after the positive edge of the clock (which is the one that is needed). To help with debugging, the lower case letters show the state just before the positive edge.

Note: `word_part` should only be `1` if `char` is a word-part char and a word has already started. Notice that at index 10 the arriving character is an `n`, which is a word-part character. But because it was not preceded by a non-word-part character a word does not start at index 10 (nor 9).

Notice that `L` is updated as each character arrives, while `N` and `A` only update when the word ends.

The testbench will trace the first few lines, and then only show trace lines when there are errors (along with a few trace lines preceding the error). For lines with an error the correct output is also shown:

```
       W-M    I  Text---->!         SPE L  N A
Trace  2-5    5 "    I II I"    __e SP_ 1  2 1
Trace  2-5    6 "   I II II"    sp_ _P_ 2  2 1
Trace  2-5    7 "  I II III"    _p_ _P_ 3  2 1
Trace  2-5    8 " I II III "    _p_ __E 3  3 3 <- Error  Correct -> __E 3  3 2
       W-M    I  Text---->!         SPE L  N A                       SPE L  N A
```

In the example above, the running average, `A`, is wrong. The module output is 3 but the testbench expects a 2.

Reset Behavior

If input `reset` is `1` on a positive edge then `len_word`, `num_words`, and `len_avg` should all be set to zero and input `char` should be considered a non-word character (regardless of its value). The trace below shows an example of reset behavior. The reset occurs at index 6. Because of when the reset occurs `bee`, rather than being a three-letter word is considered a one-letter word, the last `e`. Notice also that the average length (column `A`) does not show a value until two complete words arrive.

```
       W-M    I  Text---->!         SPE L  N A {D}
Trace  2-5    3 "      A or"    sp_ _P_ 2  1 0 {1}
Trace  2-5    4 "     A or "    _p_ __E 2  2 1 {0}
Trace  2-5    5 "    A or b"    __e SP_ 1  2 1 {1}
Trace  2-5    6 "   A or be"  R sp_ ___ 0  0 0 {1}
Trace  2-5    7 "  A or bee"    ___ SP_ 1  0 0 {1}
Trace  2-5    8 " A or bee "    sp_ __E 1  1 0 {0}
Trace  2-5    9 "A or bee k"    __e SP_ 1  1 0 {1}
Trace  2-5   10 " or bee kn"    sp_ _P_ 2  1 0 {1}
Trace  2-5   11 "or bee kno"    _p_ _P_ 3  1 0 {1}
Trace  2-5   12 "r bee knot"    _p_ _P_ 4  1 0 {1}
Trace  2-5   13 " bee knot "    _p_ __E 4  2 2 {0}
Trace  2-5   14 "bee knot  "    __e ___ 4  2 2 {0}
```

Testbench Information

The testbench will instantiate and test `word_count` at three different sizes, varying both the value of `n_avg_of` and the maximum word size. The values of `n_avg_of` will be 2, 1, and 9. To change these sizes search for `pset` in `hw04.v`. Several items in the testbench can be changed to facilitate debugging and familiarization. Search for `HW04` and read the comments for more info. The testbench will start streaming characters from the string `test_one`, and after that will construct a stream of random characters. Feel free to change `test_one` to facilitate debugging.

The testbench shows the first few errors encountered, and then silently tallies errors. After each instantiation is tested a summary of errors is shown:

```
Trace  9-7   10 " or bee   "    ___ ___ 3  3 0 {0}
Trace  9-7   11 "or bee    "    ___ ___ 3  3 0 {0}
Done with n_avg_of=9, max wd len=7. Errors: st 0, pa 0, en 0, nc 0, nw 0, av 0
```

The line starting `Done` shows a tally of errors by type after the word `Errors`. Six types of errors are tallied (all have zero errors in the output above). They are `st`, the `word_start` output, `pa`, the `word_part` output, `en`, the `word_ended` output, `nc`, the `len_word` output, `nw`, the `num_words` output, and `av`, the `len_avg` output. Remember that the line describes one instantiation, so there should be three lines printed.

The trace can be helpful for looking at values of objects in your module (not just inputs and outputs). As an example, the trace shows the value of object `char_az`, but feel free to change that or add others. To do so search for `wd_cnt.char_az`. It appears as an argument to `$sformatf` which prepares part of the trace text. Here `wd_cnt` is the instance name that the testbench uses for `word_count`. Change or add arguments to `$sformatf` to examine additional objects in your module. Be sure to change the format string to match the arguments. The end of the format string, the part in curly braces, handles the last argument `wd_cnt.char_az`.

The value of `wc` will always be chosen so that output `len_chars` never overflows. It is unlikely but not impossible that the number of words is too large for `wn`.

Word Rules

A character is an 8-bit quantity. A character is called a *word-start character* if it an ASCII alphabetic character (upper or lower case). In `word_char` net `char_wd_start` is set to one if the `char` input is a word-start character. A character is called a *word-part character* if it an ASCII alphabetic character (upper or lower case), a digit, or an underscore character. The `word_count` module net `char_wd_part` is set to one if the `char` input is a word-part character. Note that all word-start characters are word-part characters.

A word starts when the current character is a word-start character and the previous character was not a word-part character or if the module was reset in the previous cycle. A word ends when an arriving character is not a word-part character.

The length of a word is the number of characters. The output `len_word` should only be zero after a reset and until the next word starts.

Design Requirements and Goals

As always, avoid costly designs. Pay particular attention to the logic computing `len_avg`. Do not use `n_avg_of-1` adders to compute this. And definitely don't use `n_avg_of` division units.

The design can use procedural code, but it must be synthesizable. Use command `genus -files syn.tcl` to synthesis. Timing and area (cost) reports will be placed in a file named `syn-report.log`.

LSU EE 4755 Homework 5 Due: 22 November 2022

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/2022/hw05.v.html`.

Problem 0: Following instructions at `https://www.ece.lsu.edu/koppel/v/proc.html`, set up your class account, copy the assignment, and run the Verilog simulator and synthesis program on the unmodified homework file, `hw05.v`. Do this early enough so that minor problems (*e.g.*, password doesn't work) are minor problems.

Assignment Background

As we should know the synthesis program, given a Verilog description of a module, writes a design file with an optimized version of the module mapped to the chosen technology. For this assignment the chosen technology is the same Oklahoma University ASIC process we've been using throughout the semester.

An important skill for those writing Verilog descriptions is to estimate the cost and performance of those synthesized modules. In this assignment we'll look at how well the synthesis program handles the different modules we considered for computing the floating-point expression $v_0^2 + v_0 v_1 + v_1^2$. We will consider the combinational, sequential, and pipelined modules covered in class.

A synthesis script will be used to synthesize these modules, plus three arithmetic unit modules, plus additional modules created for the solution to this problem. To complete the assignment the output of the script must be understood and the synthesis script must be modified. The output of the synthesis script is similar to the output of the scripts used in prior assignments, so it should be familiar. Modifying the script will be something new, and might be a challenge for some of you. It is okay to seek help modifying the script from classmates and others, though the solutions to the problems themselves must be completed individually.

Modules

This assignment includes modules for the combinational, sequential, and pipelined implementations of the multi-step computation. They are named `ms_comb`, `ms_seq`, and `ms_pipe`. For comparison the assignment also includes modules containing a single floating-point unit, they are named `try_mult`, `try_add`, and `try_sq` (square).

Four additional modules are provided for experimentation, `m1_func`, `m1_comb`, `m1_seq`, and `m1_pipe`. These modules initially perform the computation $v_0 + v_0 v_1 + v_1^2$, but they can be modified to perform other computations. Module `m1_func` is used by the testbench to obtain a correct value, so modify it first so that it computes the desired computation. Then modify the others that you want to synthesize. (The synthesis program does not care whether a module passes the testbench, but no conclusion can be drawn from the area and delay of module that does not work correctly.)

All of these modules have the same parameters and ports, though not every module uses every port. For example, only `ms_seq` and `ms_pipe` are sequential so that the `clk` and `reset` ports on the others serve no function. These unused ports will be eliminated during optimization so they won't affect cost or timing.

Module Parameters and Floating Point Format

The modules used in this assignment all have the same parameters, these parameters specify the floating-point number format to be used. The first parameter, `wsig`, specifies the number of bits in the significand (fractional part) of the floating point number. The default value is 23, which is the same as an IEEE 754 single (C `float`). The second parameter, `wexp`, is the number of bits in the exponent. The default value is 8, which matches an IEEE single. The third parameter, `ieee`, specifies whether the IEEE floating-point format should be strictly followed. The default value

is 1, which means yes; a 0 means that special cases do not have to be handled correctly. These include `NaN` (not a number) and subnormal values. The size of the floating point number using these parameters is `1+wexp+wsig`, the extra `1` is for the sign bit.

For this assignment all modules are instantiated with `ieee=0`. This is done to explore the fuller range of optimization possibilities and also to reduce the time needed for synthesis.

The sample synthesis runs consider two formats, IEEE single in which `wsig=23` and `wexp=8`, and the ML-friendly BF16 (informally known as brain float) in which `wsig=7` and `wexp=8`. The advantage of BF16 for machine learning is that it is half the size of a single, and with a 7-bit significand, requires half the energy for multiplication than the older 16-bit FP16 format. For us the big advantage is that it takes less time to synthesize than a single.

Testbench

The testbench exercises the six modules, `ms_comb`, `ms_seq`, `ms_pipe`, `m1_comb`, `m1_seq`, and `m1_pipe` instantiated with a significand size of 7 and 23. They should all initially pass. As with other testbenchs in this class, a line will be printed for the first few module errors, and a tally will be provided for each module and size. The testbench uses `ms_func` to determine the correct output of the `ms` modules and `m1_func` to determine the correct output of the `m1` modules. When modifying the `m1` modules be sure to also modify `m1_func` so that the testbench can show you whether your modified modules do what you think they are doing.

The Synthesis Script

As with past assignments, the modules in the assignment file should be synthesized using the script `syn.tcl`. Unlike other assignments, this script will have to be modified.

The synthesis script itself is written in TCL (Tool Control Language, the abbreviation is pronounced tickle) a scripting language chosen by Cadence for scripting their EDA software. (Nowadays Python would be used. If it were up to me it would be Perl. But it's TCL.) Documentation for TCL can be found at `https://tmml.sourceforge.net/doc/tcl/`. This describes TCL, not the functionality needed to run Genus or other tools. For Genus-specific commands see the synthesis documentation linked to `https://www.ece.lsu.edu/koppel/v/ref.html`.

For this assignment it should not be necessary to use new Genus commands, just to change which modules are synthesized and which parameters to instantiate with. For that, one needs only a rudimentary knowledge of TCL, perhaps what can be learned just by looking at `syn.tcl`.

The synthesis script starts by setting some script variables, using the TCL `set` command, and by setting Genus attributes, using the Genus `set_db` command:

```
set verilog_source hw05.v
set syn_level "high"
set spew_file "spew.log"
set report_file "syn-report.log"
set_db syn_global_effort $syn_level
set rpt_chan [open $report_file w]
puts "Synthesizing at effort level \"$syn_level\"\n"
```

As one might guess `syn_level` is the amount of effort used for synthesis. Possible values are `none`, `low`, `medium`, and `high`. These initial lines are followed by the definition of a TCL procedure `syn_mod`, which emits the commands needed to synthesize a module, followed by commands to retrieve the area and delay of the synthesized module. A line of text is written showing the area and delay. It should not be necessary to modify `syn_mod` for this assignment.

Module `syn_mod` is called in a loop nest near the end of the file:

```
# List of combinational modules.
```

```
set mods_comb { ms_comb try_mult try_add try_sq }
set delay_targets { 100 0.1 }
set mods { try_mult try_add try_sq }
set mods { ms_comb ms_seq ms_pipe try_mult try_add try_sq }
set wsigs { 7 14 23 }

foreach delay_target $delay_targets {
    foreach ws $wsigs {
        foreach mod $mods {
            syn_mod $mod $delay_target " $ws 8 0 "
        }
    }
}
```

The loop nest above synthesizes each of the modules listed in `mods` (that's the inner loop). Each of these six modules is synthesized for each significand size found in `wsigs`. These modules are synthesized with each delay constraint in `delay_target`. For the code above there would be a total of $2 \times 6 \times 3$ synthesis runs. That would probably take hours.

The first `set` line writes variable `mods_comb` with a list of combinational modules. This variable must be updated with any new combinational modules that you use. Variable `mods` is set twice, first to a list of the arithmetic modules, then those are replaced with a list of the arithmetic modules and our multi-step modules. (Because of the second assignment the first assignment has no effect.) If one wanted to only synthesize the arithmetic modules one would comment out the second `mods` line. There is no need to use a loop nest. It is possible to write a `syn_mod` call for each synthesis, for example:

```
set delay_targets { 100 }
set wsigs { 7 14 23 }

syn_mod try_mult 5 "7 8 0"
syn_mod try_mult 5 "7 6 0"

# Exit before the loop nest.
close $rpt_chan
quit
foreach delay_target $delay_targets {
```

The example above does two synthesis runs. The `5` is the delay target and the quoted part are the parameters. (The parameters must be quoted so that they are read as a single argument to `syn_mod`.) In the example above, `try_mult` is synthesized with two exponent sizes, 8 bits and 6 bits, both are synthesized with a delay target of 5 ns.

To synthesize a new module (for example, one you wrote) add the name to one of the `mod` lists, or just use the name on a direct call to `syn_mod` as in the example above. **Iff the module is combinational** add the module to `mods_comb`. Not adding a combinational module to `mods_comb` will result in an error. Adding a sequential module to `mods_comb` will result in incorrect timing.

Synthesis Script Output

The synthesis script `syn.tcl` is run using the command `genus -files syn.tcl`. The run starts with a substantial amount of header output, including warnings, copyright information, and system information. Some is shown below:

```
[cyc.ece.lsu.edu] % genus -files syn.tcl
```

```
2022/11/13 16:52:05 WARNING This OS does not appear to be a Cadence supported Linux configuration.
2022/11/13 16:52:05 For more info, please run CheckSysConf in <cdsRoot/tools.lnx86/bin/checkSysConf <productId>
TMPDIR is being set to /tmp/genus_temp_566634_cyc.ece.lsu.edu_koppel_nvftYI
Cadence Genus(TM) Synthesis Solution.
Copyright 2022 Cadence Design Systems, Inc. All rights reserved worldwide.
Cadence and the Cadence logo are registered trademarks and Genus is a trademark
of Cadence Design Systems, Inc. in the United States and other countries.

[16:52:12.338826] Configured Lic search path (21.01-s002): /apps/linux/cadence/share/license/license.dat:/opt/pgi/license.dat
```

The output of the script proper (as opposed to Genus, the synthesis program) starts with an announcement of the synthesis effort level followed by a table of synthesis results:

```
Synthesizing at effort level "high"

Module Name                          Area   Delay   Delay     Synth
                                            Actual  Target     Time
ms_comb_wsig7_wexp8_ieee0          600190  12.219     0.1 ns    423 s
ms_seq_wsig7_wexp8_ieee0           445400   5.754     0.1 ns    236 s
ms_pipe_wsig7_wexp8_ieee0          797327   5.678     0.1 ns    309 s
ms_comb_wsig14_wexp8_ieee0        1363980  14.391     0.1 ns    707 s
```

Each line of the table shows the result of one synthesis run. The `Module Name` column shows the name of the module followed by the parameter values used in its instantiation. In the sample above three different modules are synthesized, `ms_comb`, `ms_seq`, and `ms_pipe`. Module `ms_comb` is synthesized once with significand of 7 bits and once with a significand of 14 bits.

The `Area` *column* shows the area given by the Genus `report area` command. The units are relative to the OSU technology. *The* `Delay Actual` *column* shows the length of critical path through the module in units of nanoseconds. *The* `Delay Target` *column* shows the delay constraint that the synthesis program was set to meet. In the example above the constraint is 0.1 ns, which means the critical path can be no longer than 0.1 ns. This constraint was intentionally set to an impossibly low value, to determine the minimum delay that the synthesis program could achieve. Normally the delay constraint is set to something achievable, perhaps 4 ns in the example above, and the synthesis program would generate the least expensive design that meets the delay constraint. *The* `Synth Time` *column* shows the wall-clock (elapsed) time needed to perform the synthesis. The wall-clock time is shown to help plan the synthesis runs, it does not directly affect or describe the design itself.

Problem 1: In class we considered three ways of implementing `multi_step`, the modules that computed $v_0^2 + v_0 v_1 + v_1^2$: A combinational version, a sequential version, and a pipelined version. Appearing below are the results from synthesizing these three modules, named `ms_comb`, `ms_seq`, and `ms_pipe`, followed by results of synthesizing modules consisting only of the Chipware floating-point multiplier, adder, and a multiplier with the same value used for both operands. These are synthesized with a large delay constraint, meaning that the cost has been minimized.

Module Name	Area	Delay Actual	Delay Target	Synth Time
ms_comb_wsig23_wexp8_ieee0	1597692	75.142	100.0 ns	229 s
ms_seq_wsig23_wexp8_ieee0	945919	29.324	100.0 ns	111 s
ms_pipe_wsig23_wexp8_ieee0	1866509	28.273	100.0 ns	205 s
try_mult_wsig23_wexp8_ieee0	525991	28.231	100.0 ns	62 s
try_add_wsig23_wexp8_ieee0	339036	27.396	100.0 ns	53 s
try_sq_wsig23_wexp8_ieee0	297753	25.504	100.0 ns	38 s
ms_comb_wsig7_wexp8_ieee0	375767	34.708	100.0 ns	75 s
ms_seq_wsig7_wexp8_ieee0	275858	15.305	100.0 ns	34 s
ms_pipe_wsig7_wexp8_ieee0	526000	14.466	100.0 ns	62 s
try_mult_wsig7_wexp8_ieee0	94274	9.346	100.0 ns	13 s
try_add_wsig7_wexp8_ieee0	140221	14.196	100.0 ns	21 s
try_sq_wsig7_wexp8_ieee0	57802	6.085	100.0 ns	8 s

(*a*) Based on the data above, show the latency and throughput of each module for the 23-bit significand. It might be necessary to look at the module descriptions (Verilog code) to answer this question.

(*b*) For each of the two significand sizes, show that the delay of the three `ms` modules are what one would expect given the delays of the three arithmetic modules.

(*c*) Using the cost of the arithmetic units, show that the cost of `ms_comb` is lower than expected, but the cost of `ms_seq` and `ms_pipe` are about or perhaps a little more than what one would expect.

Problem 2: It is welcome that the cost of `ms_comb` is lower than what one would expect based on the cost of the arithmetic units. There are several possible reasons for this, for example the synthesis program may be simplifying the two adders used in computations such as $a + b + c$ or it may be sharing hardware used to process the common b operand in expressions like $a \times b$ and $b \times c$, or perhaps it may even be transforming $v_0^2 + v_0 v_1 + v_1^2$ into $(v_0 + v_1)^2 - v_0 v_1$. Or maybe the costs for the arithmetic units shown in the table are higher than they should be.

Perform a set of synthesis runs to provide evidence for a reason that `ms_comb` cost less than its constituent parts. Consider the possible reasons given above, or one of your own. These synthesis runs can operate on one of the existing modules, a slightly modified version of the modules, or something wholly different. The modules `m1_comb`, `m1_seq`, `m1_pipe` can be used for experimentation. See the Modules section above.

Describe the results of these experiments and how they convincingly support a particular reason for the lower cost. Data from a single synthesis run, or a series of very similar runs will not be considered convincing.

The Verilog file for this assignment will be collected, but submit the answers to this question on paper or by E-mail. Please E-mail PDF files. Sending word processor source files as a final product is unprofessional, even if they are TEX files.

In your writeup:

- Indicate how you believe the synthesis program is optimizing `ms_comb`.
- Describe the modules you synthesized to come to this conclusion, and the results of synthesis. Most credit will be given for this part of the assignment.
- Explain why your experiments show that the lower cost was not due to other optimizations.

4 Fall 2021

LSU EE 4755 Homework 1 Due: 24 September 2021

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/2021/hw01.v.html`.

Problem 0: Following instructions at `https://www.ece.lsu.edu/koppel/v/proc.html`, set up your class account, copy the assignment, and run the Verilog simulator and synthesis program on the unmodified homework file, `hw01.v`. Do this early enough so that minor problems (*e.g.*, password doesn't work) are minor problems.

Problem 1: The partially completed `insert_at` module below and in the homework assignment file has three inputs, a `wa`-bit input `ia`, a `wb`-bit input `ib`, and a $\lceil \lg(\texttt{wa}+1) \rceil$-bit input `pos`, and there is one output, a `wa+wb`-bit output `o`. Complete the module following the coding requirements given further below so that `o` consists of the bits of `ia` with `ib` inserted at `pos`. That is, `o[pos-1:0]` should be set to `ia[pos-1:0]`, `o[wb+pos-1:pos]` should be set to `ib`, and `o[wa+wb-1:wb+pos]` should be set to `ia[wa-1:pos]`.

For example, let `wa=6` and `wb=2`, `ia=111111`, `ib=00`, and `pos = 2`. Then `o=11110011`. For `pos=5`, `o=10011111`. For those still not 100% sure of what `o` should be set to should look at how `o_shadow` is computed in the `testbench` module. Also, the testbench will show what the output should be when it isn't.

```
module insert_at
  #( int wa = 20, wb = 10, wo = wa+wb, walg = $clog2(wa+1) )
   ( output logic [wo-1:0] o,
     input uwire [wa-1:0] ia, input uwire [wb-1:0] ib,
     input uwire [walg-1:0] pos );

   // The line assigning mask_low must be replaced with a mask module.
   uwire [wo-1:0] mask_low = ( 1 << pos ) - 1;  // REPLACE ME!

   uwire [wo-1:0] ib_at_pos;
   shift_left #(wb,wo,walg) sl1( ib_at_pos, ib, pos );

   assign o = ia & mask_low | ib_at_pos;
endmodule
```

The `insert_at` module must be synthesizable and must not use procedural code and must not use shift operators. (That includes the line assigning `mask_low`, it must be replaced.) Instead, rely on instantiations of the provided shift and mask modules.

The testbench will test your module and report the first few errors. For example, here is the testbench output for the unmodified module:

```
Error for ia=11111111  ib=000  pos= 0  00000000000 != 11111111000 (correct)
Error for ia=11111111  ib=000  pos= 1  00000000001 != 11111110001 (correct)
Error for ia=11111111  ib=000  pos= 2  00000000011 != 11111100011 (correct)
Error for ia=11111111  ib=000  pos= 3  00000000111 != 11111000111 (correct)
Error for ia=11111111  ib=000  pos= 4  00000001111 != 11110001111 (correct)
Done with 27 tests, 15 errors found.
```

The text `00000001111 != 11110001111 (correct)` shows the output of `insert_at` to the left of the `!=` and the correct answer to the right. So in this case `00000001111` is the module output

and 11110001111 is what the module output should have been. Only the first few errors are shown, but the total number of errors is reported at the end, 15 in this case.

Synthesizability can be checked by running the synthesis script using the command **genus -files syn.tcl**. If the module is synthesizable (though not necessarily correct) a table of area and delay will be shown, for example:

```
Module Name                              Area   Delay   Delay
                                               Actual  Target
insert_at                               51832   0.987   1.000 ns
insert_at_1                             97968   0.616   0.100 ns
Normal exit.
```

One common problem encountered by beginners is setting the correct port sizes. For example, the **shift_left** module the port sizes are **wi**, **wo**, and **wolg**:

```
module insert_at #( int wa = 20, wb = 10, wo = wa+wb, walg = $clog2(wa+1) )
   ( output logic [wo-1:0] o,
     input uwire [wa-1:0] ia, input uwire [wb-1:0] ib,
     input uwire [walg-1:0] pos );
   uwire [wo-1:0] ib_at_pos;
   shift_left #(wb,wo,walg) sl1( ib_at_pos, ib, pos );
```

So the first connection to a **shift_left** instantiation must be **wi** bits, the second must be **wo** bits, and the third **wolg** bits. In the unmodified **insert_at** these parameters to **insert_at** were set explicitly to match the connection sizes. Sometimes it may be necessary to use an intermediate object or to cast in order to get the correct connection size. For example, if we wanted to shift by **pos+1** the following would not work:

```
   shift_left #(wb,wo,walg) sl1( ib_at_pos, ib, pos + 1 );
```

because the 1 in the **pos+1** expression implicitly expands it to 32 bits. (This results in a warning, but it's not good to clutter compiler output with ignorable warnings.) The problem can be solved using a cast:

```
   shift_left #(wb,wo,walg) sl1( ib_at_pos, ib, walg'(pos + 1) );
```

LSU EE 4755 Homework 2 Due: 11 October 2021

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/2021/hw02.v.html`.

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

Background

The flurry of activity machine learning is due to the success of deep neural networks (DNNs) in providing much improved solutions to otherwise hard-to-tackle problems such as natural language translation and image recognition. Deep neural network consists of multiple layers (more than two or three, otherwise they would not be deep). A *fully connected* layer computes matrix/vector products. The matrix coefficients are called *weights*, and in typical computations there are a large number of weights, so many that performance is limited by the time needed to move them around. Normally with n_i input neurons and n_o output neurons, there would be $n_i n_o$ weights, one for each input/output pair. One way to reduce the number of weights is to not require a weight for each input/output pair. In trained networks many weights are close to zero, so their removal ought to have little effect. If inference hardware (the hardware that computes the output of a layer) supports *sparse* weights then the network can be trained taking into account that some weights will be zero.

Sparsity is easier said than done because it makes the task of moving inputs and their weights to a functional unit (a multiply/add unit) more difficult. One way of lessening the difficulty is limiting which weights can be set to zero. NVidia Volta-generation GPUs support sparsity in which each group of four inputs used to compute one output is limited to two non-zero weights. Two inputs will go unused for that output (but may be used for others.)

In this assignment a module for sparse computation will be completed, `nn_sparse`. Like the Nvidia design it will operate on four inputs. But unlike the Nvidia design it can operate in both sparse and dense modes, determined by a `fmt` input. In dense mode there are four weights, but those weights have a very low precision. In sparse mode there are two weights with higher precision.

There are two challenges. One is a Verilog coding issue: instantiating an `nn2` module (see problem description) for the sparse case, and connecting it to the correct inputs, and making sure the `nn2` output reaches the module output. The other challenge is to do this in a way that maintains high performance. That is, the wider multipliers used for the sparse case will take more time and so we want to take care to not increase the critical path more than is necessary.

Testbench Output

The testbench will instantiate the `nn_sparse` module with several different parameter sets. It will then present dense and sparse patterns and check for the correct outputs. In the unmodified code all of the dense patterns should pass but nearly all of the sparse patterns should fail.

The testbench will show details on the first four errors for each configuration, followed by a tally of the total. Here is a sample showing the last error and the tally:

```
Error tn=4 for fmt 0101  084cca0 =  4.7993 !=  4.4000 (correct)
      1.0000 2.0000 + 1.2000 2.0000
      2.0000      + 2.4000
      acc1 = 0806640 = 2.1997
Done with ex6,ac18,in12,wd3 5000 tests, 2555, 0  sp, den errors found.
For ex6,ac18,in12,wd3  max diff 21132739836.039532, 0.097594  sp, den.
```

Here is what is shown for each reported error: `tn` gives a test number, `fmt` shows the value of the `fmt` input. Note that `fmt[0]` is the least significant big. After the format the error line shows the output value in hexadecimal and decimal. In the sample above they are `084cca0 = 4.7993`. After that the correct (or at least what the testbench assumes is correct) value is shown: `4.4000 (correct)`. The next line shows the expression to be computed, which will consist of four terms if the error is for a dense calculation and two terms for a sparse calculation. The example above is for a sparse calculation. The next line, `2.0000 + 2.4000`, shows the products.

Finally, the value of an object in the module is shown, `acc1`. It is shown in hexadecimal, and in decimal. Note that `acc1` is floating point, so that the hexadecimal value will let you see the sign, exponent, and significand. In most cases though, it will let you see if the value has any `x` or `z` bits.

You are encouraged to add code at this point to print out values of other signals in your module. The code to do that is:

```
// Feel free to modify or add to this to help with your solution.
$write( "      acc1 = %h = %.4f\n",
          nnsp.acc1, conv#(wexp,wsig_ac)::ftor(nnsp.acc1));
```

The `nn_sparse` instance is named `nnsp`, so `nnsp.acc1` refers to an object in the module. The function `conv#(wexp,wsig_ac)::ftor(X)` converts `X` from a floating-point format with exponent length `wexp` and significand length `wsig_ac` into a `real`. The code for this function is in `hw02.v`. Any object could be named, but remember to adjust the `$write` for data type, and the parameters to `conv` if necessary.

To aid in debugging the testbench starts out with sparse patterns in which only one weight is 1 and the others are zero. It will then use weights of 2, 0.1, 10.1. It will repeat the pattern again with two non-zero weights. After that it will use randomly chosen weights and formats. Feel free to modify the testbench to aid in your debugging. Keep in mind that the ta-bot won't test your module using the testbench in your file so removing the tests that your module fails won't help.

Synthesis Script

The synthesis script will synthesize the module at two different target delays. It takes a significant amount of time to run, so only one set of parameters is included. Feel free to modify the script, `syn.tcl` to add other sets.

Problem 1: Module `nn_sparse`, has one w_o-bit output, `o`, four w_i-bit inputs, `i[0]` to `i[3]`, a w_w-bit input, `w`, and a four-bit input, `fmt`. Input `w` can carry either two or four values, called *weights*. If `fmt=4'b1111` then `w` carries four weights, each $w_w/4$ bits. These are called *dense* weights. Otherwise `w` carries two weights, each $w_w/2$ bits, called *sparse* weights. To help get started quickly the module assigns the dense weights to four-element net `wd`.

The module is to compute `o` in one of two possible ways, depending on the value of `fmt`. When `fmt=4'b1111` the module computes `o` using the dense weights and all four values of `i`: $o = i_0w_0 + i_1w_1 + i_2w_2 + i_3w_3$, where i_0 and w_0 are values of `i[0]` and `wd[0]`. The Verilog code to do this is already in the module.

The module should work for six additional values of `fmt`: `4'b0011`, `4'b0110`, `4'b1100`, `4'b1010`, `4'b0101`, and `4'b1001`, these will be referred to as the *sparse formats*. For each of these the module should set the output to $o = i_aW_0 + i_bW_1$, where W_0 and W_1 are the two sparse weights and where a is the position of the rightmost (least significant) `1` in `fmt` and b is the position of the leftmost (most significant) `1` in `fmt`. For example, if `fmt=4'b0011` then $a = 0$ and $b = 1$ and the hardware should compute $o = i_0W_0 + i_1W_1$, and if `fmt=4'b1010` then $a = 1$ and $b = 3$ and the hardware $o = i_1W_0 + i_3W_1$.

All values are floating-point. They share a common exponent, specified by parameter `wexp`. The width of the significand of the output is specified by parameter `sig_ac`, the width of the

significand of the inputs is specified by `wsig_in`, and the width of the significand of the dense weights is specified by parameter `wsig_wd`. The layout follows IEEE 754: The most significant bit is a sign bit, that is followed by the exponent, and that is followed by the significand. So the total size of the output is `1+wexp+wsic_ac`.

To compute the dense output `nn_sparse` instantiates three modules: two `nn2` modules and `fp_add`. The `nn2` module computes $i_0 w_0 + i_1 w_1$. The `nn2` module instantiates two `hy_mult` and one `fp_add` (both described below). Details on the `nn2`, including parameters, can be learned by inspecting the module (it is in the homework file).

The `fp_add` module is a convenience wrapper around the Chipware `CW_fp_add` module.

Module `hy_mult` wraps `CW_fp_mult`, but it provides functionality that you'd think would be part of the Chipware library. Unlike the Chipware module, `hy_mult` can be instantiated so that the multiplier, multiplicand, and product each have different significand sizes, though they all share the same exponent size. (The `hy` is for hybrid, referring to the different sizes.) The module instantiates the Chipware module using the product significand size. It then widens (or shrinks) the significands of the multiplier and multiplicand inputs (called `a` and `b`). The inputs are widened by placing zeros in the least significant bits of the widened significands. This was done with the hope that the synthesis program, when performing optimization, would see that these bits were zero and so optimize away the affected partial products. Experiments using Genus (version 211) confirmed that optimization was occurring.

(*a*) The table below shows synthesis script output for the hybrid multiplier at a variety of sizes. Based on this table there is a good and bad way to connect the hybrid multiplier. Design your module taking this data into account. In the table the parameter values are concatenated with the module name, and numbers are added on the end to avoid duplicating a name. Remember that with a large delay target cost is the only goal, and with a 1ns goal speed is the primary goal.

For purposes of interpreting the data below, assume your design will be instantiated with parameters `{wexp 5} {wsig_ac 14} {wsig_in 8} {wsig_wd 4}`. (These can be found in the synthesis script.)

Module Name	Area	Delay Actual	Delay Target
hy_mult_wsig_a5_wsig_b5_wsig_p20	62541	7.466	100.000 ns
hy_mult_wsig_a10_wsig_b10_wsig_p20	131839	12.799	100.000 ns
hy_mult_wsig_a10_wsig_b5_wsig_p20	84546	10.636	100.000 ns
hy_mult_wsig_a5_wsig_b10_wsig_p20	92111	9.851	100.000 ns
hy_mult_wsig_a15_wsig_b5_wsig_p20	108593	13.440	100.000 ns
hy_mult_wsig_a5_wsig_b15_wsig_p20	123209	12.643	100.000 ns
hy_mult_wsig_a4_wsig_b8_wsig_p14	71354	8.435	100.000 ns
hy_mult_wsig_a8_wsig_b4_wsig_p14	63890	9.007	100.000 ns
hy_mult_wsig_a14_wsig_b8_wsig_p14	131244	12.047	100.000 ns
hy_mult_wsig_a8_wsig_b14_wsig_p14	144388	11.824	100.000 ns
hy_mult_wsig_a3_wsig_b7_wsig_p12	59985	7.737	100.000 ns
hy_mult_wsig_a7_wsig_b3_wsig_p12	53501	8.081	100.000 ns
hy_mult_wsig_a12_wsig_b7_wsig_p12	110260	12.113	100.000 ns
hy_mult_wsig_a7_wsig_b12_wsig_p12	117097	11.660	100.000 ns
hy_mult_wsig_a5_wsig_b5_wsig_p20_22	130160	2.398	1.000 ns
hy_mult_wsig_a10_wsig_b10_wsig_p20_22	324729	3.046	1.000 ns
hy_mult_wsig_a10_wsig_b5_wsig_p20_22	189191	2.690	1.000 ns
hy_mult_wsig_a5_wsig_b10_wsig_p20_22	214533	2.684	1.000 ns
hy_mult_wsig_a15_wsig_b5_wsig_p20_22	248189	2.742	1.000 ns

```
hy_mult_wsig_a5_wsig_b15_wsig_p20_22    302877   2.900   1.000 ns
hy_mult_wsig_a4_wsig_b8_wsig_p14_22     171041   2.369   1.000 ns
hy_mult_wsig_a8_wsig_b4_wsig_p14_22     135568   2.232   1.000 ns
hy_mult_wsig_a14_wsig_b8_wsig_p14_22    296160   3.030   1.000 ns
hy_mult_wsig_a8_wsig_b14_wsig_p14_22    321123   3.232   1.000 ns
hy_mult_wsig_a3_wsig_b7_wsig_p12_22     127217   2.308   1.000 ns
hy_mult_wsig_a7_wsig_b3_wsig_p12_22     132936   1.994   1.000 ns
hy_mult_wsig_a12_wsig_b7_wsig_p12_22    263353   2.823   1.000 ns
hy_mult_wsig_a7_wsig_b12_wsig_p12_22    260279   2.951   1.000 ns
```

(*b*) Modify `nn_sparse` so that it computes the correct outputs for both sparse and dense inputs, and is coded for higher speed. Since a sparse weight is larger than a dense weight a multiplier designed to use sparse weights would cost more and take more time than one designed for dense weights. But, when computing sparse weights only one addition operation is needed. Design your module so that this benefit is realized.

Modify only `nn_sparse` to solve the problem, and use the provided FP units. (Contact me if you feel modifying other modules is needed. (Note that you are free to modify the testbench and related files to help with debugging. But the solution itself should only involve changes to `nn_sparse`.)

Solving this problem requires good debugging skills. Use SimVision (see the course procedures page) to view what is going on inside your module. Also take advantage of the testbench output, and don't hesitate to modify it so that it provides tests that will help you better understand your module.

LSU EE 4755 Homework 3 Due: 18 October 2021

To help solve the problems below, look at problems listed in the simple model slides, 2020 Homework 4, 2019 Midterm Exam Problem 2b and c, and especially 2018 Final Exam problems 1 and 2.

Problem 1: As requested in the subproblems below use the simple model to determine the cost and delay of the `insert_at` module from the solution to Homework 1 (see last page) instantiated with `wa` $= w_a$ and `wb` $= w_b$, and using $C_{\text{lsb}}(w_a)$ for the cost of the `mask_lsb` module and $D_{\text{lsb}}(w_a)$ for the delay of the `mask_lsb` module. The `wo` and `walg` parameters are not set so you can use their default values, $w_o = w_a + w_b$, $l_a = \lceil \lg(w_a + 1) \rceil$, and $l_b = \lceil \lg w_b \rceil$, in your answers.

For partial credit, and to help you solve the problems provide a sketch of the inferred hardware. It may help to first solve the problem for specific values of w_a and w_b, and then to generalize for arbitrary w_a and w_b.

(a) Find the cost and delay of the hardware inferred for the line of Verilog from `insert_at` shown below. Just for the hardware described on the line. There's no trick, this part is easy. Just remember to express your answers in terms of w_a, w_b, and w_o.

```
assign o = ia_high | ib_at_pos | ia_low;
```

(b) Find the cost and delay of the `shift_left` module instances `slc` and `slb` taking into account any constant inputs and assuming that the synthesis program infers a logarithmic shifter. Don't forget that your answer must be in terms of w_a, w_b, w_o, l_a, and l_b, and that these denote the parameters of `insert_at`, not the parameters of the shifters. For more information on the logarithmic shifter see the additional material provided for the Set 1 lectures on the course lectures page.

Before cutting-and-pasting simple-model cost and delay expressions for a logarithmic shifter, take a close look at the parameters set for `slc` and `slb` and be sure to optimize for them. Notice that unlike typical shifters, the shift-out and shift-in ports are not the same size and that the shift amount is not necessarily ceiling-log-two of the input width.

Hint: The cost and delay for one of these shifters will be really easy to compute.

(c) Find the cost and delay of `insert_at`. Use the answers above and work out cost and delay for the remaining hardware in the module. Don't forget to use $C_{\text{lsb}}(w_a)$ for the cost of the `mask_lsb` module and $D_{\text{lsb}}(w_a)$ for the delay of the `mask_lsb` module.

Problem 2: Some of you may have seen this coming: Find expressions for $C_{\text{lsb}}(w)$, the cost of the `mask_lsb` module and $D_{\text{lsb}}(w)$, the delay of the `mask_lsb` module, in both cases `wo` $= w$, where `wo` is the parameter used in the `mask_lsb` definition. Assume a well-optimized design, not something that uses w $\lceil \lg w \rceil$-bit magnitude comparison units.

Hint: Think about the problem for about 30 minutes, then look at 2018 Final Exam Problems 1 and 2.

An uncommented Homework 1 solution appears below.
For the full version visit `https://www.ece.lsu.edu/koppel/v/2021/hw01-sol.v.html`.

```
module insert_at
  #( int wa = 20, wb = 10, wo = wa+wb, walg = $clog2(wa+1) )
   ( output logic [wo-1:0] o,
     input uwire [wa-1:0] ia,
     input uwire [wb-1:0] ib,
     input uwire [walg-1:0] pos );

   uwire [wa-1:0] mask_low;
   mask_lsb #(wa) ml(mask_low, pos);
   uwire [wa-1:0] ia_low = ia & mask_low;
   uwire [wa-1:0] ia_high_low = ia & ~mask_low;

   localparam int wblg = $clog2(wb);
   uwire [wo-1:0] ia_high;
   shift_left #(wa,wo,wblg) slc( ia_high, ia_high_low, wblg'(wb) );

   uwire [wo-1:0] ib_at_pos;
   shift_left #(wb,wo,walg) slb( ib_at_pos, ib, pos );

   assign o = ia_high | ib_at_pos | ia_low;

endmodule

module shift_left
  #( int wi = 4, wo = wi, wolg = $clog2(wo) )
   ( output uwire [wo-1:0] o,
     input uwire [wi-1:0] i,
     input uwire [wolg-1:0] amt );
   assign o = i << amt;
endmodule

module mask_lsb
  #( int wo = 6, wp = $clog2(wo+1) )
   ( output logic [wo-1:0] o, input uwire [wp-1:0] n1 );
   always_comb for ( int i=0; i<wo; i++ ) o[i] = i < n1;
endmodule
```

LSU EE 4755 Homework 4 Due: 11 November 2021

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/2021/hw04.v.html`.

Problem 0: If necessary, follow the instructions at `https://www.ece.lsu.edu/koppel/v/proc.html` to set up your class account, copy the assignment, and run the Verilog simulator and synthesis program on the unmodified homework file, `hw04.v`. Do this early enough so that minor problems (*e.g.*, password doesn't work) are minor problems.

Teamwork

Students can work on this assignment in teams. Each student should submit his or her own assignment but list team members. It is recommended that one team member be responsible for learning SimVision.

Every member of a team that has completed a project, must be capable of re-solving the problem. It is recommended that all team members re-solve the problem on their own for their own pedagogical benefit.

Problem 1: Module `bit_keeper` has a w_b-bit output `bits` ($_b$ is for width of buffer) and a 1-bit output `ready`. Think of output `bits` as a long bit vector (w_b bits long) that is edited using the module's inputs. Commands to edit `bits` are given using four-bit input `cmd` (command), w_i-bit input `din` (data in), and w_s-bit input `pos` (position). The module is to operate sequentially using input `clk`.

Complete `bit_keeper` as described below, and make sure that it is synthesizable. As always, code should be written clearly, and designs should not be costly or slow.

When completed `bit_keeper` should operate as follows. On a positive edge of `clk` action is taken based on the value of `cmd`. The possible values of `cmd` are: `Cmd_Reset`, `Cmd_None`, `Cmd_Write`, and `Cmd_Rot_To`. (These can be used as constants in your code. The constants are defined by `enum Command`.) Some commands will be complete in one cycle (the cycle in which the `cmd` is set up to the positive edge of `clk`). Other commands will take multiple cycles.

Be sure to understand the details of how multi-cycle commands execute. When a multi-cycle command starts the `ready` output must be set to zero and must be held at zero until the command completes. The command and its arguments will only be held at the inputs **for one cycle**, and so at the next positive clock edge they will be gone. The `cmd` input will be set to `Cmd_Nop`, and the `pos` and `din` inputs will be set to random values. This means that the inputs of multi-cycle commands that will be needed in subsequent cycles must be saved in registers.

The testbench can emit a trace of commands and their effects. This trace is used below to illustrate what the module is supposed to do. The trace is collected after the command completes. A trace entry starts with the word `Cycle`. The cycle number is shown, followed by command details, followed by the state of bits.

For `Cmd_Reset` output `bits` should be set to zero. Also, any internal registers should be set to zero. The command should complete at the positive edge. This should set `ready` to 1. In the trace below the reset command set bits back to zero. Notice that the command completes in one cycle (based on the cycle numbers).

```
Cycle 307 -- test 73: Cmd_Nop                    : bits = 01401f4
Cycle 308 -- test 74: Cmd_Reset                  : bits = 0000000
```

For `Cmd_Rot_To` the value in `bits` must be rotated so that the contents of `bits[0]` is moved to `bits[pos]`, `bits[1]` is moved to `bits[(pos+1)%wb]`, and so on. This is like a left shift of `pos` bits, except that the most significant `pos` bits of `bits` are rotated into the the `pos` least significant bits. In the trace below the rotate command rotates four bits (one hexadecimal digit). Notice that the most-significant digit on the first line is rotated to the least significant digit after the rotation command.

```
Cycle 301 -- test 71: Cmd_Nop                    : bits = 401401f
Cycle 306 -- test 72: Cmd_Rot_To pos 4           : bits = 01401f4
```

This rotation **must be performed** using two instances of module `rot_left`. One instance should rotate by 1, the other rotates by a larger value, call it r_b, of your choosing. Each clock cycle the value of `bits` is rotated using one of these, but never both in the same clock cycle. Use the r_b-bit rotate instance until the number of bit positions to shift is $\leq r_b$, then use the 1-bit rotate instance.

Command `Cmd_Write` has two forms based on the value of input `pos`. If `pos` is zero then the least significant w_b bits of `bits` should be written with `din`. This should complete at the positive edge. Otherwise, bits `pos` through `pos+wi-1` of `bits` should be written with `din`—but not directly. Instead, `bits` should be rotated so that bit `pos` is at the least-significant position, then the data should be written, then `bits` should be rotated back to its original position. Use only the two `rot_left` instances.

The trace below shows a write with `pos=0`:

```
Cycle 417 -- test 86: Cmd_Nop                    : bits = 0000240000
Cycle 418 -- test 87: Cmd_Write  pos 0, data 7   : bits = 0000240007
```

When `pos` is non-zero the writes take longer:

```
Cycle  96 -- test 20: Cmd_Nop                    : bits = 0a0000003c
Cycle 107 -- test 21: Cmd_Write  pos 27, data 4  : bits = 0a2000003c
```

No action is needed for command `Cmd_Nop`. In fact, this is the command that will be present while the external hardware, including the testbench, is waiting for other commands to complete.

The testbench will test `bit_keeper` at two sizes. At each size detailed information is given for the first few errors. That includes a trace of commands leading up to the error, followed by the erroneous command, and what the `bits` should have been. After each error the testbench sets its shadow value of `bits` to the erroneous output so that subsequent tests can pass. Here is in example of the output:

```
Cycle  22 -- test  0: Cmd_Rot_To pos 20          : bits = 0000000000
Cycle  54 -- test  1: Cmd_Rot_To pos 31          : bits = 0000000000
Cycle  55 -- test  2: Cmd_Nop                    : bits = 0000000000
Cycle  96 -- test  3: Cmd_Write  pos 37, data 2  : bits = 4000000000
Cycle  97 -- test  4: Cmd_Nop                    : bits = 4000000000
Cycle 103 -- test  5: Cmd_Rot_To pos 5           : bits = 0000000008
Cycle 104 -- test  6: Cmd_Write  pos 0, data 3   : bits = 0000000003
Error in test  7: Cmd_Write  pos 1, data 2   : 0000000c04 != 0000000005 (correct)
```

For multi-cycle commands the testbench will wait for `ready` to go to zero and then back to one. If that does not happen after a certain number of cycles the testbench will *timeout*, meaning that it will give up waiting and print a `CYCLE LIMIT EXCEEDED` message. If there is a timeout while a command is in progress (meaning that `ready` did go to zero, but did not return to one) the testbench will show a trace of recent history, followed by an indication of what it was waiting for:

```
Exit from clock loop at cycle 16000, limit 16000, ** CYCLE LIMIT EXCEEDED **
```

```
** Preceding Commands **
Cycle   7 -- test  0: Cmd_Rot_To pos 20         : bits = 0000000000
Cycle  14 -- test  1: Cmd_Rot_To pos 31         : bits = 0000000000
Cycle  15 -- test  2: Cmd_Nop                   : bits = 0000000000

** In-Progress Command **
test  3: Cmd_Write  pos 37, data 2
 -- Awaiting ready = 1.
```

If the testbench does not timeout then it will print a tally of the number of errors after testing each `bit_keeper` instance. Also, as a measure of quality, the testbench reports the average number of cycles to perform `Cmd_Rot_To` and `Cmd_Write` (with non-zero `pos`). For example,

```
Starting tests for (wb=40,wi=4)
Finished 200 tests for (wb=40,wi=4), 0 errors.
Avg cyc  Cmd_Rot_To 5.5 (67)  Cmd_Write 10.6 (35)

Starting tests for (wb=28,wi=8)
Finished 140 tests for (wb=28,wi=8), 0 errors.
Avg cyc  Cmd_Rot_To 4.2 (57)  Cmd_Write 8.2 (18)
```

The lines starting `Avg cyc` report timing. The number in parentheses is the number of times the command was issued. So for the first set of tests `Cmd_Rot_To` was tried 67 times, and the average number of cycles taken to complete it was 5.5.

A lower number for `Avg cyc` can indicate a good design, or that certain rules were not followed.

It is very important that debugging tools are used. Take advantage of the testbench messages to see what is going wrong. Run SimVision to get a detailed look at what your module is doing.

LSU EE 4755 Homework 5 Due: 17 November 2021

Problem 1: Solve 2020 Solve-Home Final Exam Problem 1, which asks for the inferred hardware for the $v_0^2 + v_0v_1 + v_1^2$ module that we covered in class. For those who may have forgotten how to use a pencil, or never learned, an SVG version of the illustration is available at `https://www.ece.lsu.edu/koppel/v/2020/fe-ms.svg`. Use Inkscape or your favorite SVG editor on the file.

Problem 2: This assignment does not have a Problem 2. I know that's confusing but the alternative is also confusing.

Problem 3: Solve 2020 Solve-Home Final Exam Problem 3, which asks for a timing analysis of the $v_0^2 + v_0v_1 + v_1^2$ module. An SVG version of the diagram is at `https://www.ece.lsu.edu/koppel/v/2020/fe-ms-t.svg`.

LSU EE 4755 Homework 6 Due: 22 November 2021

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/2021/hw06.v.html`.

Problem 0: If necessary, follow the instructions at `https://www.ece.lsu.edu/koppel/v/proc.html` to set up your class account, copy the assignment, and run the Verilog simulator and synthesis program on the unmodified homework file, `hw06.v`. Do this early enough so that minor problems (*e.g.*, password doesn't work) are minor problems.

Teamwork

Students can work on this assignment in teams. Each student should submit his or her own assignment but list team members. It is recommended that one team member be responsible for learning SimVision.

Every member of a team that has completed a project, must be capable of re-solving the problem. It is recommended that all team members re-solve the problem on their own for their own pedagogical benefit.

Problem 1: Complete module `multi_step_pipe` so that it is a pipelined version of the `multi_step_functional` or `multi_step_seq` modules. All of modules are in `hw06.v`. (This is based on 2020 Solve-Home Final Exam Problem 2.)

The module must accept a new set of `v0` and `v1` values each clock cycle and produce a new result each clock cycle. In the module set `nstages` to the number of stages in your module, so that the value of output `result` is based on the inputs that appeared `nstages` clock cycles ago.

Instantiate as many Chipware floating-point multiplication and addition modules as needed. (Do not use procedural code for the arithmetic.) The critical path should pass through at most one floating-point module.

Also, set the `ready` output at the correct time. Output `ready` should be set to the value that `start` has `nstages` ago.

The testbench will show a trace for about the first three computations (inputs in which `start` was 1), and will show a trace for the ten cycles preceding each error, up to seven errors. A tally of errors will be shown at the end. Here is a sample of the testbench for a working module:

```
MS Pipe  Cyc  20  In:   0.0,  0.0 ->   0.0  Rdy 0 , Res:   0.0
MS Pipe  Cyc  21  start=0                  Rdy 0 , Res:   0.0
MS Pipe  Cyc  22  start=0                  Rdy 0 , Res:   0.0
MS Pipe  Cyc  23  start=0                  Rdy 1 , Res:   0.0 Good
MS Pipe  Cyc  24  start=0                  Rdy 0 , Res:   0.0
MS Pipe  Cyc  25  In:   1.0,  0.0 ->   1.0  Rdy 0 , Res:   0.0
MS Pipe  Cyc  26  start=0                  Rdy 0 , Res:   0.0
MS Pipe  Cyc  27  start=0                  Rdy 0 , Res:   0.0
MS Pipe  Cyc  28  In:   0.0,  1.0 ->   1.0  Rdy 1 , Res:   1.0 Good
MS Pipe  Cyc  29  start=0                  Rdy 0 , Res:   0.0
MS Pipe  Cyc  30  start=0                  Rdy 0 , Res:   0.0
MS Pipe  Cyc  31  start=0                  Rdy 1 , Res:   1.0 Good
For MS Pipe ran 400 tests: Errors: 0 wrong val, 0 bad timing
```

On a cycle in which input `start` is `1` the trace line will show the word `In:` followed by the values of `v0` and `v1`, and to the right of `->` the correct result (which should appear `nstages` cycles later). The text to the right of `Rdy` shows the value of the `ready` output. If the value is incorrect it is followed by an `x`, for example, `Rdy 1x,`.

The text to the right of `Res:` shows the value on the module `result` output. That is followed by text commenting on the result. A comment will be shown if `Rdy` is `1` or if an output is expected. `Good` indicates a correct value at the correct time. `XX: Need Rdy` indicates that the correct value appears at the correct time, but the `ready` output isn't `1`. `XX: Wrong` indicates the wrong value at the time when an output was expected. `XX: Early` indicates the correct value arriving too early. `XX: Unexpected` indicates the wrong value at a time when no value at all is expected.

Below are excerpts from the testbench output on the unmodified module.

```
MS Pipe  Cyc  20  In:   0.0,  0.0 ->   0.0  Rdy 0X, Res:   0.0 XX: Need Rdy
MS Pipe  Cyc  21  start=0                   Rdy 1X, Res:   0.0 XX: Early
MS Pipe  Cyc  22  start=0                   Rdy 0 , Res:   0.0
MS Pipe  Cyc  23  start=0                   Rdy 0 , Res:   0.0
MS Pipe  Cyc  24  start=0                   Rdy 0 , Res:   0.0
MS Pipe  Cyc  25  In:   1.0,  0.0 ->   1.0  Rdy 0X, Res:   0.0 XX: Wrong
MS Pipe  Cyc  26  start=0                   Rdy 1X, Res:   1.0 XX: Unexpected
MS Pipe  Cyc  27  start=0                   Rdy 0 , Res:   0.0
MS Pipe  Cyc  28  In:   0.0,  1.0 ->   1.0  Rdy 0X, Res:   0.0 XX: Wrong
MS Pipe  Cyc  29  start=0                   Rdy 1X, Res:   0.0 XX: Early
MS Pipe  Cyc  30  start=0                   Rdy 0 , Res:   0.0
MS Pipe  Cyc  31  start=0                   Rdy 0 , Res:   0.0
MS Pipe  Cyc  32  start=0                   Rdy 0 , Res:   0.0
MS Pipe  Cyc  33  In:   1.0,  1.0 ->   3.0  Rdy 0X, Res:   0.0 XX: Wrong
MS Pipe  Cyc  34  start=0                   Rdy 1X, Res:   1.0 XX: Unexpected
MS Pipe  Cyc  35  In:  -8.6,  5.0 ->  55.9  Rdy 0X, Res:   0.0 XX: Wrong
MS Pipe  test 4: Inputs at cyc 35, result expected at cyc 35. Wrong val: h'00000022
0.0000 != 55.8659 (correct)
MS Pipe  Cyc  36  In:   0.4,  3.9 ->  16.7  Rdy 1 , Res:  -8.6 XX: Wrong
MS Pipe  test 5: Inputs at cyc 36, result expected at cyc 36. Wrong val: h'c109657e
-8.5873 != 16.7235 (correct)
```

The following is the testbench output on a module in which `nstages` is set too low by 1, and in which `v00` is used where `v01` should be:

```
MS Pipe  Cyc  20  In:   0.0,  0.0 ->   0.0  Rdy 0 , Res:   0.0
MS Pipe  Cyc  21  start=0                   Rdy 0 , Res:   0.0
MS Pipe  Cyc  22  start=0                   Rdy 0X, Res:   0.0 XX: Need Rdy
MS Pipe  Cyc  23  start=0                   Rdy 1X, Res:   0.0 XX: Early
MS Pipe  Cyc  24  start=0                   Rdy 0 , Res:   0.0
MS Pipe  Cyc  25  In:   1.0,  0.0 ->   1.0  Rdy 0 , Res:   0.0
MS Pipe  Cyc  26  start=0                   Rdy 0 , Res:   0.0
MS Pipe  Cyc  27  start=0                   Rdy 0X, Res:   0.0 XX: Wrong
MS Pipe  Cyc  28  In:   0.0,  1.0 ->   1.0  Rdy 1X, Res:   2.0 XX: Unexpected
MS Pipe  Cyc  29  start=0                   Rdy 0 , Res:   0.0
MS Pipe  Cyc  30  start=0                   Rdy 0X, Res:   0.0 XX: Wrong
MS Pipe  test 2: Inputs at cyc 28, result expected at cyc 30. Wrong val: h'00000000
0.0000 !=  1.0000 (correct)
MS Pipe  Cyc  31  start=0                   Rdy 1X, Res:   1.0 XX: Unexpected
```

```
MS Pipe  Cyc  32  start=0                    Rdy 0 , Res:   0.0
MS Pipe  Cyc  33  In:   1.0,  1.0 ->   3.0  Rdy 0 , Res:   0.0
MS Pipe  Cyc  34  start=0                    Rdy 0 , Res:   0.0
MS Pipe  Cyc  35  In:  -8.6,  5.0 ->  55.9  Rdy 0X, Res:   0.0 XX: Wrong
MS Pipe  Cyc  36  In:   0.4,  3.9 ->  16.7  Rdy 1X, Res:   3.0 XX: Unexpected
MS Pipe  Cyc  37  In:  -9.5, -4.5 -> 152.0  Rdy 0X, Res:   0.0 XX: Wrong
```

Make sure that your modules are synthesizable.

The smart way to solve the problem is to base the design on `ms_functional`. Remember that the control logic in `multi_step_seq`, such as logic related to `step`, is not needed in a pipelined implementation. The solution should be relatively short and uncomplicated. For example, no conditionals are needed.

A good way to start is to compute everything in one stage, and when that's correct break the logic into stages so that the critical path passes through at most one floating-point module.

5 Fall 2020

LSU EE 4755 Homework 1 Due: 16 September 2020

☣ *Paper copies will not be accepted. E-mail your solution to* `koppel@ece.lsu.edu`. *A single PDF file is preferred.*

Problem 1: In the Module-Port-versus-Module-Parameter section of lecture code `https://www.ece.lsu.edu/koppel/v/2020/l005-review.v.html` there are several module designs for computing $c_1x + c_2y$, where c_1 and c_2 are constants and x and y are module inputs. The point of that section and of the modules was to illustrate the SystemVerilog differences between module parameters and ports (syntax issues, for example) and also how they relate to the hardware being modeled.

(*a*) Draw a diagram of module `c1x_c2y_good`, shown below, using its default parameter values (which are different than the ones in the lecture code). Show the contents of all instantiated modules and appropriately label ports and wires. (See 2016 Homework 1 Problem 3 for a diagram showing instantiated modules. Also see module `arb_exp` and the illustration that follows in `https://www.ece.lsu.edu/koppel/v/2020/l015-syn-comb-str.v.html`.)

- Use the default parameter values of the module `c1x_c2y_good` shown below.
- Use the appropriate parameter values for the `mult_by_c` instances. *Hint: appropriate is not a synonym for default.*
- Show the ports for all modules.
- Show the number of bits in each wire.
- Label wires with the symbols used below (such as `p1` and `prod`) and take care to place the label on the correct side of a module boundary. (In the `two_pie` illustration from `https://www.ece.lsu.edu/koppel/v/2020/l005-review.v.html` look at the wire carrying labels `x`, `i1`, and `a`.).

```
module mult_by_c
  #( int w = 8, int c = 16, int w2 = w+$clog2(c) )
   ( output uwire signed [w2-1:0] prod, input uwire signed [w-1:0] a );
   assign prod = a * c;
endmodule

module c1x_c2y_good
  #( int c1 = 4, int c2 = 7, int w = 15,
     int w2 = w + $clog2(c1) + $clog2(c2) )
   ( output logic signed [w2-1:0] s,   input uwire signed [w-1:0] x, y );

   uwire [w2-1:0] p1, p2;

   mult_by_c #(w,c1,w2) m1(p1,x);
   mult_by_c #(w,c2,w2) m2(p2,y);

   assign  s = p1 + p2;

endmodule
```

(*b*) Draw a diagram of module `c1x_c2y_okay` below using its default parameter values (which are different than the defaults used in the lecture code). Show the same details, such as ports, as was requested for the previous part.

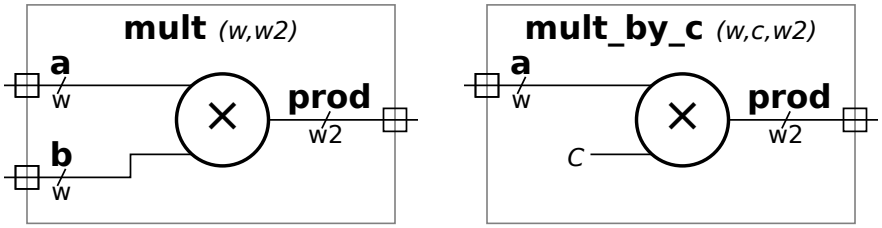
```
module mult
  #( int w = 8, int w2 = 2 * w )
   ( output uwire signed [w2-1:0] prod, input uwire signed [w-1:0] a, b );
   assign prod = a * b;
endmodule

module c1x_c2y_okay
  #( int c1 = 4, int c2 = 7, int w = 15,
     int w2 = w + $clog2(c1) + $clog2(c2) )
   ( output logic signed [w2-1:0] s,   input uwire signed [w-1:0] x, y );

   uwire [w2-1:0] p1, p2;
   uwire [w:1] C1 = c1, C2 = c2;  // Convert constants to desired size.

   mult #(w,w2) m1(p1, x, C1);
   mult #(w,w2) m2(p2, y, C2);

   assign  s = p1 + p2;

endmodule
```

Problem 2: Synthesis programs optimize a design to minimize cost while meeting timing constraints. The illustration below for the `mult` and `mult_by_c` modules (used in the slides) show how the multiplier can be simplified when one of the inputs is a convenient constant, 1.

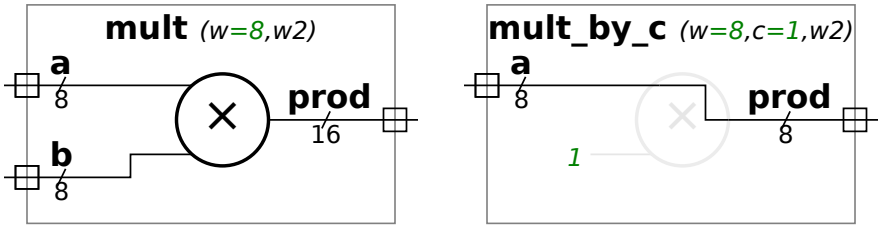
Show how the `c1x_c2y_good` module from the first problem can be optimized based on the default `c1=4` and `c2=7` values. To do so show the multiplier replaced by much simpler hardware, such as adder(s). A correct solution uses only one adder for both multipliers, bit relabeling, *plus the adder used to combine* `p1` *and* `p2`.

Note: As originally assigned, and until Tuesday, 15 September 2020 at about 16:15, the problem stated that a correct solution uses only one adder, implying but not specifically stating that the one adder was the replacement for the multipliers and that there would also be and adder computing `p1+p2`, for a total of two adders.

Before instantiation and optimization.



After instantiation and optimization.



LSU EE 4755 Homework 2 Due: 18 September 2020

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/2020/hw02.v.html`.

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

Problem 1: Module `nn4x4`, below, has two inputs, a 4-element vector `ai` and a 4×4 matrix `wht`, and one output, a 4-element vector `ao`. Output `ao` is set to the product of `wht` and `ai`. Parameter `ww` (width of weight) gives the number of bits in the elements `wht` and parameter `wa` (width of activation) gives the number of bits in the elements of `ai` and `ao`.

The illustration below the module shows hardware that might be inferred for `nn4x4`. The illustration also includes three dotted green boxes. These are suggestions on how to hierarchically decompose this large, some would say unwieldy, module.

The two smaller boxes, labeled `nn1x2b`, show hardware computing part of one output using two inputs. The larger box, labeled `nn1x4b` shows hardware computing one output using four inputs. As those who took the time to look at the illustration might have guessed by now the module suggested by `nn1x4b` can be constructed using two instances of `nn1x2b`. Further, a `nn4x4b` can be constructed using four instances of `nn1x4b`. Sounds interesting? Good!

The homework file `hw02.v` contains module `nn4x4`, it is there for your reference. The file also contains mostly empty modules `nn4x4b`, `nn1x4b`, and `nn1x2b`. Complete these so that they compute the same output as `nn4x4` and are constructed as suggested in the illustration and follow the guidelines below.

Module `nn4x4b` must instantiate exactly four `nn1x4b` modules and `nn1x4b` must instantiate exactly two `nn1x2b` modules. Module `nn1x4b` will also need an adder. Module `nn4x4b` has parameters. Don't change them. The other modules should have similar parameters with the same default values as `nn4x4b`. Do not ignore the parameters when declaring inputs and outputs. A standing rule in this class is that all code must be clearly written.

The modules must be synthesizable. This should not be a challenge for this assignment. Verify synthesizablity by running the synthesis script using the command `genus -files syn.tcl`.

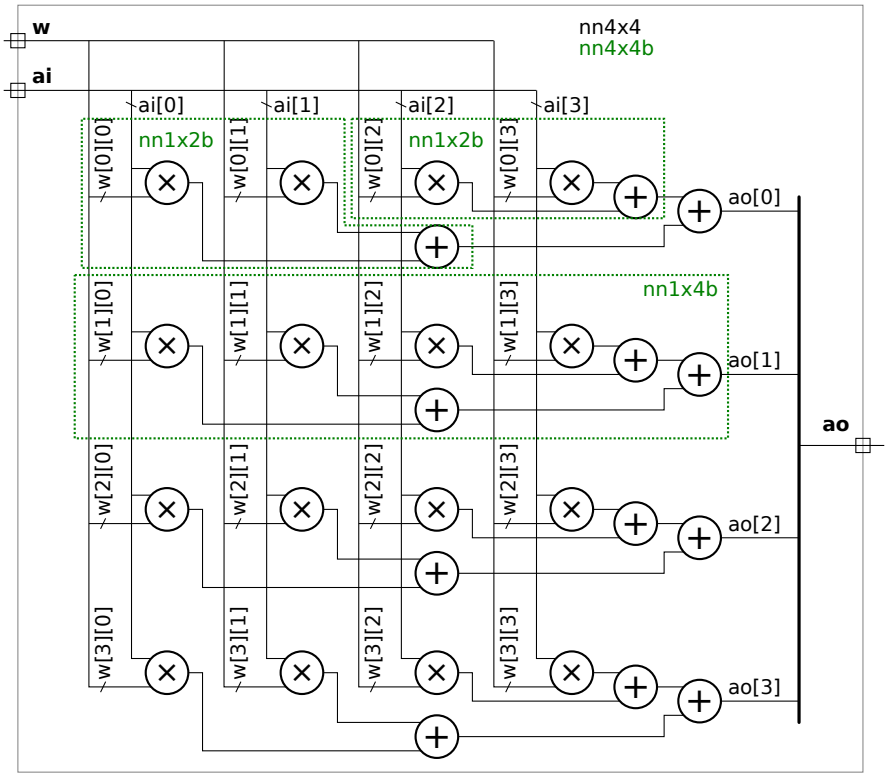
Those who fear they might forget to address some part of the problem described here can rest easy. There is a checklist in the part of the Verilog file where the solution goes.

To help you solve the problem in stages the testbench will perform three rounds of tests. In the first round, labeled `n12`, only output `ao[0]` will be examined and only inputs `ai[0]` and `ai[1]` will have non-zero values. In the second round, labeled `n14`, only output `ao[0]` will be examined but all inputs will have non-zero values. The full test, all outputs checked and all inputs are non-zero, is labeled `n44`.

Some might find it helpful to look at two past homework assignments in which a flat module was to be decomposed hierarchically. The simpler one (perhaps) is 2019 Homework 1, in which a multiplier is decomposed. But the multiplier had two scalar inputs, `a` and `b`. In this (2020) assignment one input is a 1-D array (`ai`) and the other is a 2-D array (`wht`). In the Fall 2017 Homework 1 Problem 2 an 8-input multiplexor is to be decomposed. The mux input `a` is a 1-D array that had to be split between two instances.

Module and illustration on the next page.

```
module nn4x4
  #( int wa = 10, ww = 5 )
   ( output uwire [wa-1:0] ao[4],
     input uwire [wa-1:0] ai[4],
     input uwire [ww-1:0] wht[4][4] );

   assign ao[0] = ai[0] * wht[0][0] + ai[1] * wht[0][1]
          + ai[2] * wht[0][2] + ai[3] * wht[0][3];

   assign ao[1] = ai[0] * wht[1][0] + ai[1] * wht[1][1]
          + ai[2] * wht[1][2] + ai[3] * wht[1][3];

   assign ao[2] = ai[0] * wht[2][0] + ai[1] * wht[2][1]
          + ai[2] * wht[2][2] + ai[3] * wht[2][3];

   assign ao[3] = ai[0] * wht[3][0] + ai[1] * wht[3][1]
          + ai[2] * wht[3][2] + ai[3] * wht[3][3];

endmodule
```



LSU EE 4755 Homework 3 Due: 13 October 2020

The deadline has been extended by one day, to 13 October (late at night) due to power outages caused by Hurricane Delta.

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/2020/hw03.v.html`.

Problem 0: Following instructions at `https://www.ece.lsu.edu/koppel/v/proc.html`, set up your class account (if you haven't already), copy the assignment, and run the Verilog simulator and synthesis program on the unmodified homework file, `hw03.v`.

Homework Overview and Neural Network Background

The goal of Homework 2 was to describe a 4×4 matrix/vector multiply circuit hierarchically. That goal is generalized here where an $n_i \times n_o$ matrix is multiplied by an n_i-element vector. In Homework 2 each `ao[o]` was computed by a tree connection of multipliers. Here both linear and tree connections will be tried. Also, the module in this assignment will optionally do something about overflow.

The modules in this assignment and in Homework 2 could be used any place where matrix/vector multiplication is needed, but they were designed with a particular application in mind that some students might have guessed from the names used: artificial neural networks. The `nn` prefix is for neural network. The output and one input name starts with `a`, that's for *activation*, which can be though of as a neuron. The weights model connections between neurons.

A completely connected neural network layer performs a matrix vector multiplication. The multiply/add operation needed to compute that is also an important operation for other compute-intensive workloads, including graphics and many forms scientific computation. General-purpose CPUs and GPUs were designed in part to perform multiply/add operations efficiently—on some workloads, including graphics and scientific computation.

One thing that sets neural network (a technique for *machine-learning [ML]*) workloads apart is operand precision. Graphics uses 32-bit values for coordinates, many scientific computation uses 64-bit values. Lower precision would be less effective. But machine learning can get by with less precision, and with different precision for the weights than the activations. Lower precision reduces the amount of energy needed for computation (which is often a limiter), and the amount of data that needs to be moved. This is especially important for weights in fully-connected layers.

The modules in this assignment allow for different precision for inputs, outputs, and weights. When the precision of the output is low there is a danger of overflow. That is often handled by saturating a value at the maximum representable quantity.

Reference Module, `nnOxIbe`

A goal of this assignment is to write a Verilog description of a module performing the same computation as a reference module, `nnOxIbe`. Module `nnOxIbe` has two inputs, an n_i-element vector of w_i-bit integers, `ai`, and an $n_o \times n_i$ matrix of w_w-bit integers, `wht`; the module has one output, an n_o-element vector of w_o-bit integers, `ao`, where n_i, n_o, w_i, w_w, and w_o are the values of the similarly named module parameters. All integers are unsigned. Output `ao` is set to the product of matrix `wht` and column vector `ai` with overflow handled as described further below.

Most will find it easiest to inspect the code in `nnOxIbe` (below) to resolve any remaining certainty about what this module does. For the others let $r(p) = \sum_{q=0}^{n_i-1} H_{p,q} a_i(q)$, where $H_{p,q}$ is the equivalent of the Verilog `wht[p][q]`, and $a_i(q)$ is the equivalent of `ai[q]`. Then either

$a_o(p) = \min\{\, r(p),\, 2^{w_o} - 1 \,\}$ or $a_o(p)$ is set to the low w_o bits of $r(p)$, depending on the value of parameter `sat`.

Module `nn0xIbe` initially computes a 32-bit precision value (see variable `acc`) for each `ao[i]`. If `sat=0` then `ao[i]` is assigned the low `wo` bits of this value. If `sat!=0` then `ao[i]` is set to the minimum of `acc` and $2^{w_o}-1$. As some may have guessed, `sat` is short for saturating arithmetic. (In saturating arithmetic an overflow is replaced by the maximum representable value. For example, for 4-bit unsigned integers and a saturating add: $11_2 + 1110_2 = 1111_2$.)

```
module nnOxIbe
  #( int no = 4, ni = 4, wo = 10, wi = 4, ww = 5, sat = 0 )
   ( output logic [wo-1:0] ao[no],
     input uwire  [wi-1:0] ai[ni],  input uwire [ww-1:0] wht[no][ni] );

   // The maximum possible value of each element of ao.
   localparam logic [wo-1:0] smax = ~wo'(0);

   always_comb
     for ( int o = 0;  o < no;  o++ ) begin
        automatic int unsigned acc = 0;
        for ( int i=0; i<ni; i++ ) acc += ai[i] * wht[o][i];

        // If sat is non-zero replace a value that would overflow
        // ao[o] with the maximum value that ao[o] can hold.
        ao[o] = sat && acc > smax ? smax : acc;
     end

endmodule
```

Testbench

(This part is best read after looking at Problems 1 and 2.) The testbench will instantiate sixteen (as of this writing) configurations of `nn0xI`. For each configuration, three sets of tests are performed, similar to the ones performed for Homework 2. A grand total of errors is printed at the end, such as `Total number of errors:  660`. Above that the number of errors are grouped in various ways. For example:

```
All Sat 0      220 errors.
All Sat 1      220 errors.
All Sat 2      220 errors.
Linear         330 errors.
Linear Sat 0   110 errors.
Linear Sat 1   110 errors.
Linear Sat 2   110 errors.
Tree           330 errors.
Tree Sat 0     110 errors.
Tree Sat 1     110 errors.
Tree Sat 2     110 errors.
Total number of errors: 660
```

The line reading `Linear 330 errors` shows the total number of errors of all configurations for which `tr=0`. The line `Linear Sat 0 110 errors.` shows the number of errors on linear modules with `sat=0`.

Further up specific inputs and incorrect outputs are shown. For example:

```
** Starting tests for no=3, ni=5, wo=15, wi=9, ww=8, sat=2
Testing module Linear

** Starting test set n12  (1 outputs, 2 inputs) for Linear **
Error test # 0, output 0: z != 32767 (correct)
Error test # 1, output 0: z != 32767 (correct)
Error test # 2, output 0: z != 26759 (correct)
Done with 10 n12 tests on Linear: 10 errors found.
```

In the example above output `ao[0]` was `z` (unconnected) but should have been 32767.

In test set `n12` the inputs and weights are chosen so that the only non-zero output should be `ao[0]` and so that only `ai[0]` and `ai[1]` are non-zero. In set `n1*` all inputs can have non-zero values but weights are chosen so that only `ao[0]` is non-zero. In test set `n**` all inputs can be non-zero and all outputs can be non-zero.

Problem 1: Complete module `nnOxI` so that it produces the same output as `nnOxIbe` and does so using generate statements to either describe a linear or recursive module as described below.

Module `nnOxI` is to be the starting point in all cases. It has the same parameters as `nnOxIbe`, plus it also has a parameter `tr`. The solution to this problem requires modification to `nnOxI` and to module `nn1xI`. Both are in `hw03.v`.

Multiplication and addition of values should be performed by instances of the provided arithmetic modules, `nnAdd`, `nnMult`, and `nnMADD` (multiply/add). These modules can perform saturating arithmetic.

Module `nnOxI` should instantiate `no` (that's a number) `nn1xI` modules. Each `nn1xI` instance should compute one output of `nnOxI`. The `tr` parameter in `nnOxI` indicates whether each `nn1xI` should compute its output using a linear arrangement of modules or a tree arrangement.

For the linear arrangement `nn1xI` should use a generate loop to instantiate `nnMADD` modules. The critical path (without optimization) should be $O(n_i)$ multiply/add operations. For the tree arrangement `nn1xI` should either instantiate two copies of itself or for the base case, the arithmetic modules.

For an example of a module describing a linear arrangement of hardware see `min_n` in the generate/elaborate lecture code, `https://www.ece.lsu.edu/koppel/v/2020/l025-gen-elab.v.html`. For an example of a module describing a tree arrangement of hardware see `min_t` in the lecture code.

Be sure to specify the appropriate parameters when instantiating modules, including the `sat` parameter.

- Do not make ports wider than they need to be.
- Make sure that the modules pass all tests.
- Make sure that the module is synthesizable. (Use command `genus -files syn.tcl` to synthesize.) The area should be > 0.
- Code should be clearly written.

Problem 2: Module `nnOxIbe` honors the `sat` parameter after it has computed a 32-bit `ao[o]` value. (That is, it first computes a 32-bit result, then it checks if it's too large.) That's fine for software, but it would be wasteful for our hardware because we'd need to provide 32-bit precision

hardware for all arithmetic. Or is it really that wasteful? First, we don't necessarily need 32 bits. The maximum value of `ao[o]` depends on `wi`, `ww`, and `ni`, so we only need enough bits to hold that. Also, the saturating arithmetic modules may be inflating cost for two reasons: the cost of detecting and handling saturation, and the fact that algebraic optimizations are impeded when saturation is performed. So, it may be less expensive to compute a value for `ao[o]` to a precision greater than `wo`, and then just saturate that value. This way saturation is performed once per output, rather than `ni` times.

Modify your modules so that when `sat=2` saturation is performed as described above.

LSU EE 4755 Homework 4 Due: 28 October 2020

☣ *Paper copies will not be accepted. E-mail your solution to* `koppel@ece.lsu.edu`. *A single PDF file is preferred.*

This assignment refers to the solution to Homework 3. Pieces are shown below, the complete solution can be found at `https://www.ece.lsu.edu/koppel/v/2020/hw03-sol.v.html` *and in the directory where the original assignment was copied from.*

Problem 1: Using the simple model compute the cost and delay of the `nnAdd` module from Homework 3 (shown below) for both `sat=0` and `sat=1`. Do so after applying optimizations for constants. Show the cost and delay in terms of w. *Hint: See the simple model notes,* `https://www.ece.lsu.edu/koppel/v/2020/lsli-simple-model.pdf`*, for the cost of a ripple adder.*

- Show cost and delay in terms of w.
- Don't forget to optimize for constant values.
- Assume that the adder will be implemented using a ripple circuit.
- Indicate both the delay of the least-significant bit of the sum and the delay of the most significant bit of the sum. *Answering this part correctly and applying it to the other problems in this assignment will reveal something important about the impact of detecting overflow and of the different methods of doing so.*

```
module nnAdd #( int w = 5, sat = 0 )
              ( output uwire [w-1:0] so, input uwire [w-1:0] a, b );
   uwire [w:0] s = a + b;
   localparam logic [w-1:0] smax = ~w'(0);
   assign so = sat && s[w] ? smax : s[w-1:0];
endmodule
```

There are more problems on the next pages.

Problem 2: Using the simple model compute the cost and delay of the `nnMult` module from Homework 3 for `sat=1`. Let w denote the setting of both `wa` and `wb` (they are to be set to the same value), and let y denote the setting of `wp`. Solve this for $y < 2w$. Do so after applying optimizations for constants.

Solve this using the following cost for an unsigned integer multiplier with two w-bit inputs and a $2w$-bit output: the cost using the simple model is $10w^2\,\mathrm{u_c}$ and the delay is $[8w + 2]\,\mathrm{u_t}$ for the complete product and $[4i + 2]\,\mathrm{u_t}$ for bit position i. (The LSB is at position $i = 0$.) (For more details on how those were derived see the comments after the Linear Multiplier in `https://www.ece.lsu.edu/koppel/v/2020/mult-seq.v.html`.)

- Show the cost and delay in terms of w and y.
- Solve this for $y < 2w$.
- Don't forget to optimize for constant values.

```
module nnMult #( int wa = 5, wb = 6, wp = wa + wb, sat = 0 )
   ( output uwire [wp-1:0] p,  input uwire [wa-1:0] a, input uwire [wb-1:0] b );

   localparam logic [wp-1:0] pmax = ~wp'(0);
   localparam int wmx = wp > wa+wb ? wp : wa+wb;
   uwire [wmx-wp:0] phi;
   uwire [wp-1:0] plo;
   assign {phi,plo} = a * b;
   assign p = sat && wp < wa + wb && phi ? pmax : plo;

endmodule
```

There are more problems on the next pages.

Problem 3: Using the simple model determine the cost and performance of module `nn1xI` (shown on the next page) for the configurations described below. In all cases, let n denote the value of `ni`, w denote the value of `ww` and `wi` (which are the same) and y denote the value of `wo`. Assume the same hardware costs as the first two problems (modifying sizes and accounting for cascading where appropriate).

(a) Find the cost (not delay in this part) for `sat=0`, `tr=0`, and $y > 2w$ (that's one configuration) and for `sat=0`, `tr=1`, and $y > 2w$ (that's a second configuration). The two costs will be very similar.

- Show the costs in terms of n, w, and y.

(b) Find the delay (not cost in this part) for `sat=0`, `tr=0`, and $y > 2w$ (that's one configuration) and for `sat=0`, `tr=1`, and $y > 2w$ (that's a second configuration). The two delays will be very different.

- Show the delays in terms of n, w, and y.
- When computing the total delay don't forget to take into account the time that inputs arrive at each port, especially for the multiplier.
- When computing total delay account for cascading of ripple units.

(c) Find the delay for `sat=1`, `tr=0`, and $y > 2w$ (that's one configuration) and for `sat=1`, `tr=1`, and $y > 2w$ (that's a second configuration). The two delays should be very different from each other and from the delays from the previous problem.

```
module nn1xI #( int wo = 10, wi = 4, ww = 5, ni = 2, tr = 0, sat = 0 )
   ( output uwire [wo-1:0] ao,
     input uwire [wi-1:0] ai[ni],
     input uwire [ww-1:0] wht[ni] );

   if ( tr ) begin

      if ( ni == 1 ) begin

         nnMult #(wi,ww,wo,sat) mult(ao, ai[0], wht[0] );

      end else begin

         localparam int nlo = ni / 2;
         localparam int nhi = ni - nlo;
         uwire [wo-1:0] aolo, aohi;
         nn1xI #(wo,wi,ww,nlo,1,sat) nnlo(aolo, ai[0:nlo-1], wht[0:nlo-1]);
         nn1xI #(wo,wi,ww,nhi,1,sat) nnhi(aohi, ai[nlo:ni-1], wht[nlo:ni-1]);
         nnAdd #(wo,sat) add(ao,aolo,aohi);

      end

   end else begin

      uwire [wo-1:0] s[ni-1:-1];
      assign s[-1] = 0;
      assign ao = s[ni-1];

      for ( genvar i = 0;  i < ni;  i++ )
        nnMADD #(ww,wi,wo,sat) madd( s[i], wht[i], ai[i], s[i-1] );

   end

endmodule

module nnMADD #( int wa = 10, wb = 5, ws = wa + wb, sat = 0 )
   ( output uwire [ws-1:0] so,
     input uwire [wa-1:0] a, input uwire [wb-1:0] b, input uwire [ws-1:0] si);

   uwire [ws-1:0] p;
   nnMult #(wa,wb,ws,sat) mu(p, a, b);
   nnAdd #(ws,sat) ad(so, si, p);

endmodule
```

There are even more problems on the next pages.

Problem 4: Consider module `nn0xI` instantiated with `no=1`, `tr=0`, for both `sat=1` and `sat=2`. (A slightly simplified version appears below.) Let n denote the value of `ni`, w denote the value of `wi` and `ww` (which are the same), and let y denote the value of `wo`.

Assume that $2w < y < \lceil \lg n(2^w - 1)^2 \rceil$. That is, y is large enough so that the multipliers can't overflow but not so large that the adders can't overflow.

(a) Compute the cost and delay for both the `sat=1` and `sat=2` cases. For `sat=1` just re-use answers from the previous problems.

- Show answers in terms of n, w, and y.
- Don't forget that the value of `wo` in the `nn1xI` instantiations depends upon `sat`.

(b) In terms of the costs computed above is `sat=2` always better, always worse, or sometimes better than `sat=1`? Be specific of course.

```
module nnOxI #( int no = 4, ni = 2, wo = 10, wi = 4, ww = 5, tr = 0, sat = 0 )
   ( output uwire [wo-1:0] ao[no],
     input uwire [wi-1:0] ai[ni],   input uwire [ww-1:0] wht[no][ni] );

   // Compute number of bits to represent largest possible value that
   // can appear on an ao.
   localparam int wr = $clog2( ( 2**wi - 1 ) * ( 2**ww  - 1 ) * ni );

   if ( sat < 2 ) begin

      for ( genvar i = 0;  i < no;  i++ )
        nn1xI #(wo,wi,ww,ni,tr,sat) row( ao[i], ai, wht[i] );

   end else begin

      for ( genvar i = 0;  i < no;  i++ ) begin

         uwire [wr-1:0] ar;
         nn1xI #(wr,wi,ww,ni,tr,0) row( ar, ai, wht[i] );
         assign ao[i] = ar[wr-1:wo] ? ~wo'(0) : ar[wo-1:0];

      end

   end

endmodule
```

Problem 5: *Zero points will be given for the answer to this question, but please try your very best to answer it.* Suggest a method of saturating `ao` that avoids the extra `wo` bits needed (for `nn1xI`) when `sat=2` but also avoids the critical-path-killing saturation logic used when `sat=1`. Your solution could add extra ports to all modules except `nn0xI`. A correct solution would detect overflow under the same conditions as `nn0xI` does with `sat=1`.

LSU EE 4755 Homework 5 Due: 3 December 2020

☣ *Paper copies will not be accepted. E-mail your solution to* `koppel@ece.lsu.edu`. *A single PDF file is preferred.*

Problem 1: Complete 2019 Final Exam Problem 2, which asks for a timing analysis of a `best_match` module, and related questions.

Problem 2: Complete 2019 Final Exam Problem 3, in which code implementing an illustrated module is to be completed. The module computes a Fibonacci sequence. Submit your solution by E-mail, handwritten is acceptable. However, those wishing to write Verilog code and to use a testbench can copy `/home/faculty/koppel/pub/ee4755/hw/2020/hw05` to class account and solve it. Instructions for remote access are in `https://www.ece.lsu.edu/koppel/v/proc.html` (look for the "Remote Access" heading).

6 Fall 2019

LSU EE 4755 Homework 1 Due: 18 September 2019

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/2019/hw01.v.html`.

Problem 0: Following instructions at `https://www.ece.lsu.edu/koppel/v/proc.html`, set up your class account, copy the assignment, and run the Verilog simulator and synthesis program on the unmodified homework file, `hw01.v`. Do this early enough so that minor problems (*e.g.*, password doesn't work) are minor problems.

Homework Overview

In class you were told that for common operations, such as shifting, addition, and multiplication, it's better to use Verilog operators in procedural code than to re-invent the wheel by writing Verilog to implement those operations. This point was made when covering the shift module in the introductory lectures. For example, if you need a shifter it's better to just use the shift operator:

```
module shift_right_operator
  ( output uwire [15:0] shifted,
    input uwire [15:0] unshifted, input uwire [3:0] amt );
   assign shifted = unshifted >> amt;
endmodule
```

than to write code for your own shifter:

```
module shift_right_logarithmic
  ( output uwire [15:0] sh, input uwire [15:0] s0, input uwire [3:0] amt );
   uwire [15:0] s1, s2, s3;
   mux2 st0( s1, amt[0], s0, {1'b0, s0[15:1]} );
   mux2 st1( s2, amt[1], s1, {2'b0, s1[15:2]} );
   mux2 st2( s3, amt[2], s2, {4'b0, s2[15:4]} );
   mux2 st3( sh, amt[3], s3, {8'b0, s3[15:8]} );
endmodule

module mux2( output uwire [15:0] x,
             input uwire select, input uwire [15:0] a0, a1 );
    assign x = select ? a1 : a0;
endmodule
```

The reason for showing the implementation of shifters, and other common operations, was to teach general design concepts using operations that you should be familiar with. That will be the approach in this homework, in which a multiplier is to be implemented.

Testbench Code

The testbench for this assignment, which can be run when visiting the file in Emacs in a properly set-up account by pressing F9, tests the multiply modules. Modules `mult_operator` and `mult16` should pass, `mult16_tree` awaits your solution. A sample of the end of the testbench output appears below:

```
Starting testbench...
Error in mult16_tree test    0:  xxxxxxxx != 00000001 (correct)
Error in mult16_tree test    1:  xxxxxxxx != 00000002 (correct)
Error in mult16_tree test    2:  xxxxxxxx != 00000020 (correct)
Error in mult16_tree test    3:  xxxxxxxx != 00000020 (correct)
```

```
Error in mult16_tree test    4:  xxxxxxxx != 139dff24 (correct)
Error in mult16_tree test    5:  xxxxxxxx != 4839cb7b (correct)
Mut mult_operator  ,    0 errors (0.0% of tests)
Mut mult16_flat    ,    0 errors (0.0% of tests)
Mut mult16_tree    , 1000 errors (100.0% of tests)
Memory Usage - 38.6M program + 154.6M data = 193.2M total
CPU Usage - 0.0s system + 0.0s user = 0.1s total (70.4% cpu)
Simulation complete via $finish(2) at time 10 US + 0
./hw01.v:218       $finish(2);
ncsim> exit
```

A count of the number of tests and errors is shown for three modules. The testbench shows the first six errors it finds on each module. To see more than six modify the testbench (search for `err_limit`). In the output above the testbench is showing that the module outputs are `x` (uninitialized) which of course don't match the expected outputs.

Use Simvision to debug your modules. Feel free to modify the testbench so that it presents inputs that facilitate debugging.

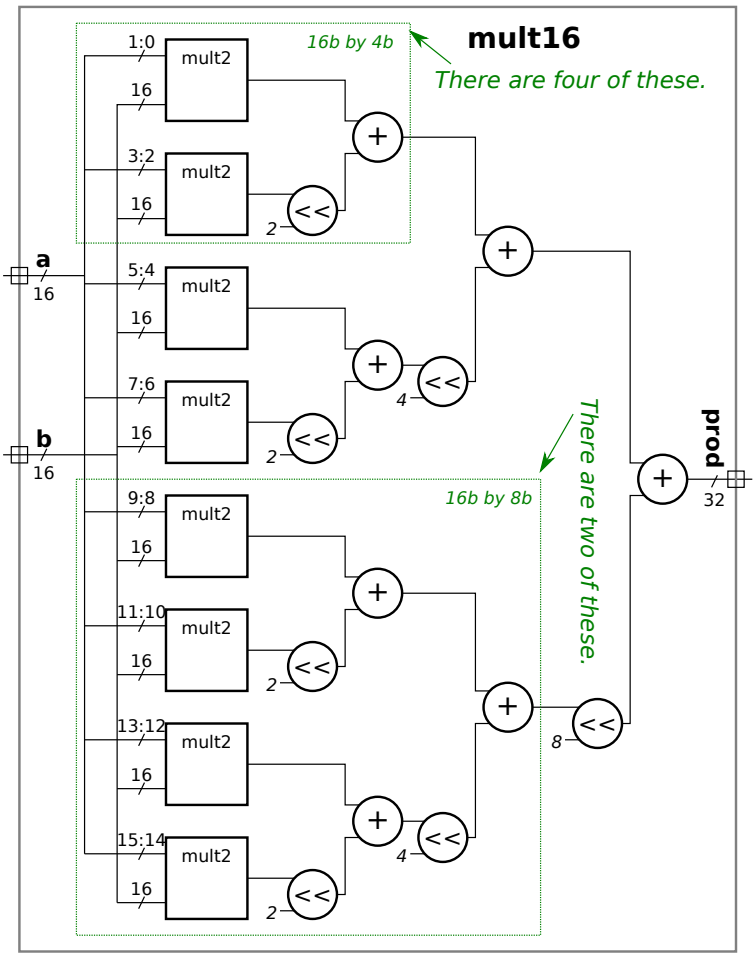
Synthesis

The synthesis script, `syn.tcl`, will synthesize the three modules each with two delay targets, an easy 10 ns and a un-achievable 0.1 ns. If the module doesn't synthesize −.001 s is shown for the delay. The script is run using the shell command `genus -files syn.tcl`, which invokes Cadence Genus.

The synthesis script shows area (cost), delay, and the delay target in a neat table. Additional output of the synthesis program is written to file `spew.log`. Sample synthesis script output appears below:

Problem 1 on next page.

Problem 1: The illustration to the right shows a sketch of a multiplier, `mult16`, with two 16-bit inputs and a 32-bit output. The multiplier is constructed from `mult2` modules, shifters (<<), and adders. The illustrated module is similar to the multiplier in `mult16_flat` in `hw01.v`. The `mult2` modules have two inputs, one is two bits, the other is 16 bits. Each input holds an unsigned integer. The output, 18 bits, is the product of the two inputs. Notice that each `mult2` module is connected to two bits of `a` and all bits of `b`. The outputs of the `mult2` modules are shifted and added together in such a way that `prod` is the correct product of `a` and `b`.



There are two parts of `mult16` surrounded by green boxes. The upper one, labeled *16b by 4b*, contains two `mult2` modules. The label is explaining that the boxed material multiplies a 16-bit number by a 4-bit number. A similar box could have been put around the next pair of `mult2` modules, etc. The hardware within each of these four boxes would be identical. (The bit slices at the upper `mult2` inputs, such as `1:0` and `5:4` are different, but that can be taken care of outside the green box.) Think about the poor soul who might have just typed in all the Verilog for `mult16` and then suddenly realizes this. All that person would have had to do would be to code one module, call it `mult4_tree`, and just instantiate it four times. Here is an almost empty version of `mult4_tree`:

```
module mult4_tree
  ( output uwire [0:0] prod, // Need to change output size.
    input uwire [3:0] a, input uwire [15:0] b );

   mult2 mlo( /* finish */ );
   mult2 mhi( /* finish */ );

endmodule
```

Alert students might suspect that we don't actually instantiate `mult4_tree` *four* times because the *16b by 8b* section itself could be a module which would contain only two instantiations of `mult4_tree`. That would be correct.

Modify modules `mult16_tree`, `mult8_tree`, and `mult4_tree` found in `hw01.v` so that they implement the multiplier described above. Module `mult16_tree` must instantiate exactly two `mult8_tree` modules, module `mult8_tree` must instantiate exactly two `mult4_tree` modules, and

`mult4_tree` must use the two `mult2` modules that are already instantiated (but with the ports missing).

In each module use implicit structural code or behavioral code to combine the outputs of that module's two instantiated modules. It might be helpful to look at `mult16_flat` for examples of instantiation and implicit procedural code.

Start with module `mult16_tree`. You can test your changes to `mult16_tree` by putting placeholder code in `mult8_tree`, such as `assign prod = a*b;`. Don't forget to change the port sizes on `mult8_tree` to what they should be based on the diagram.

Once the testbench reports zero errors move the placeholder to `mult4_tree` and complete `mult8_tree`. Continue until the three modules are finished.

Some of the port sizes are set to 1 bit, `[0:0]`. Those are placeholders, change those to the correct sizes, but no larger. Credit will be deducted for oversized ports, especially if all ports are made 32 bits.

Pay attention to port-size warnings when running the simulator.

Problem 2: The synthesis script will synthesize `mult16_tree` from Problem 1, plus two already working modules, `mult16_flat` and `mult_operator`, which just uses the multiply operator.

If the synthesis program were perfect then all three modules would have the same cost and delay because they each do exactly the same thing (multiply) and so the optimization algorithms would have found the same lowest-cost circuit from each one. Spoiler alert: Genus is not perfect.

Guess which module you think will be the fastest or least expensive, and explain why. Then run the synthesis script and comment on whether the results met your expectations.

LSU EE 4755 Homework 2 Due: 8 October 2019

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/2019/hw02.v.html`.

Problem 0: Following instructions at `https://www.ece.lsu.edu/koppel/v/proc.html`, set up your class account, copy the assignment, and run the Verilog simulator and synthesis program 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 Correction (December 2019)

When assigned in October 2019 this assignment defined clz backward, starting at the least-significant bit. That has been corrected in this version and in the posted code.

Homework Overview

A *count leading zeros (clz)* operation returns the number of consecutive zeros starting at the most significant bit of an integer's binary representation. For example, the clz of 00101_2 is 2, the clz of 101_2 is 0, and the clz of 32-bit number 0_2 is 32. The Verilog module below computes the clz of its input:

```
module clz
  #( int w = 19, int ww = $clog2(w+1) )
   ( output var logic [ww-1:0] nlz, input uwire logic [w-1:0] a );

   uwire [w:0] aa = { a, 1'b1 };
   always_comb for ( int i=0; i<=w; i++ ) if ( aa[i] ) nlz = w-i;
endmodule
```

The module was written as behavioral code, but it does turn out to be synthesizable. Nevertheless, one may wonder if the synthesis program will do a good job with this. (Later in the semester we will learn what kind of hardware will be inferred for the description above.) One way to find out is to design a module which *should* be efficient and see how well it compares to what the synthesis program does with the module above. That, and the use of generate statements, is the subject of this assignment.

Testbench Code

The testbench for this assignment, which can be run when visiting the file in Emacs in a properly set-up account by pressing F9, tests the `clz_tree` module at several different widths. All should initially fail. A shortened sample of the testbench output appears below:

```
ncsim> run
** Starting tests for width 1.
Error for width  1: input 1:  z != 0 (correct).
Error for width  1: input 0:  z != 1 (correct).
Error for width  1: input 1:  z != 0 (correct).
Error for width  1: input 0:  z != 1 (correct).
Width 1, done with 10 tests, 10 errors.
** Starting tests for width 2.
Error for width  2: input 3:  z != 0 (correct).
Width 2, done with 20 tests, 20 errors.
** Starting tests for width 5.
```

```
[snip]
Error for width 17: input 08959:   z != 0 (correct).
Width 17, done with 170 tests, 170 errors.
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
Total number of errors: 610
```

The testbench prints the details of the first four errors it finds, and after that prints just one detail time per width. A total for each width and a grand total are printed, see the transcript above.

Use Simvision to debug your modules. Feel free to modify the testbench so that it presents inputs that facilitate debugging.

Synthesis

The synthesis script, `syn.tcl`, will synthesize `clz` (for reference) and `clz_tree` (your solution). Each module will be synthesized at three widths, and with two delay targets, an easy 10 ns and a un-achievable 0.1 ns. If a module doesn't synthesize −.001 s is shown for its delay. The script is run using the shell command `genus -files syn.tcl`, which invokes Cadence Genus. If you would like to synthesize additional modules or sizes edit `syn.tcl` near the bottom.

The synthesis script shows area (cost), delay, and the delay target in a neat table. Additional output of the synthesis program is written to file `spew.log`.

Problem 1: Complete module `clz_tree` so that it computes the clz of its input in a tree-like fashion. For the non-terminal case it should instantiate two `clz_tree` modules and each should operate on part of the input, `a`. The outputs of these two modules should be appropriately combined. To help you get started, a recursive solution to Homework 1, `mult_tree`, is in `hw02.v`.

An easy mistake to make is using the wrong sized variable in a module port connection. Previously the Verilog software (`ncelab` to be precise) would issue a warning which was easy to miss. Now a port size mismatch is a fatal error.

For maximum credit do not use adders in your design. Adders can be avoided if the size of the low module is always a power of 2.

See the Verilog code check boxes for additional items to check for.

Problem 2: Run the synthesis program and indicate how your module compares to the behavioral module, `clz`. Indicate which results are expected, and which are not expected, and explain why.

LSU EE 4755 Homework 3 Due: 23 October 2019

Problem 1: Appearing below is a module excerpted from the solution to Homework 1. Compute the cost and delay of this module using the simple model under the following assumptions:

- The inputs arrive at $t = 0$. Don't assume that any bit is early or late, they all arrive at exactly $t = 0$.
- A ripple adder will be used to implement addition.
- Apply obvious optimizations. In particular, don't use a BFA if a BHA would suffice. And only use a BHA if that is needed.
- Don't overlook the fact that one of the shifter inputs is a constant.

Show the cost and delay in terms of `wa` and `wb`, but use symbol a for `wa` and b for `wb`. For example, "The cost is $(a+b)9\,\mathrm{u_c}$ and the delay is $(a+b)2\,\mathrm{u_t}$." (Those answers assume that BFAs are used for the entire module, which is wrong.)

The simple model slides (AOTW) don't show the cost and delay of a BHA, so work that out yourselves.

```
module mult_piece
  #( int wa = 16, int wb = 16, int wp = wa + wb,
     int wn = wa / 2, int wx = wb + wn )
   ( output uwire [wp:1] prod,
     input uwire [wx:1] prod_lo, prod_hi );

      assign prod = prod_lo + ( prod_hi << wn );

endmodule
```

There's another problem on the next page!

Problem 2: A w-bit multiplier needs to add together w partial products using $w-1$ adders. A naïve timing analysis of a non-tree ripple adder implementation would compute a delay of $w(2 \times 2w + 2) = (4w^2 + w)\,\mathrm{u_t}$ for the $2w$-bit product using the simple model and ignoring ripple-unit cascading. As we should know $4w^2$ is not a good term to have in an expression for time. The goal of this problem is to see how the tree multiplier compares to this naïve timing.

Appearing below is the Bonus Solution to Homework 1 in which a single `mult_tree` module is used rather than separate modules `mult16_tree`, `mult8_tree`, etc. Also shown is a module, `my_module` that instantiates the `mult_tree`. Also shown a page or two ahead is the diagram from Homework 1. You may want to use this to help work out the solution to this problem.

Analyze the cost and performance of `my_module` as described below. When computing the cost and performance don't forget to account for the full elaboration, not just the top level. For example, `my_module` with `w=4` consists of one `mult_tree` at `w=4` and two `mult_tree` modules at `w=2`, and four `mult_tree` modules at `w=1`.

```
module mult_tree
  #( int wa = 16, int wb = 16, int wp = wa + wb )
   ( output logic [wp:1] prod,
     input uwire [wa:1] a,
     input uwire [wb:1] b );

   if ( wa == 1 ) begin

      assign prod = a ? b : 0;
      // Equivalent to: prod = a * b;

   end else begin

      // Split a in half and recursively instantiate a module for each half.
      localparam int wn = wa / 2;
      localparam int wx = wb + wn;

      uwire [wx:1] prod_lo, prod_hi;

      mult_tree #(wn,wb) mlo( prod_lo, a[wn:1],    b );
      mult_tree #(wn,wb) mhi( prod_hi, a[wa:wn+1], b );

      // Combine the partial products.
      always_comb prod = prod_lo + ( prod_hi << wn );

   end
endmodule

module my_module
  #( int w = 8, int wp = 2 * w )
   ( output uwire [wp-1:0] p,
     input uwire [w-1:0] x, y );
   mult_tree #(w,w) mt1(p,x,y);
endmodule
```

(*a*) Compute the cost of `my_module` using the same assumptions as in Problem 1. The cost must

be in terms of w. It's okay, indeed encouraged, to use sample values like $w = 16$ when working out the problem, but once you have it figured out give the answer in terms of w. (If you have not solved Problem 1 then use the incorrect sample answers provided in Problem 1.)

The following identity may be helpful: $\sum_{i=0}^{m-1} 2^i = 2^m - 1$. In such a summation i might indicate the level of recursion and 2^i might indicate the number of modules at that recursion level. For the top level of the recursion $i = 0$.

(*b*) Compute the delay of the multiplier using a simplifying assumption similar to the one used in Problem 1: when computing the delay of `prod = prod_lo + ( prod_hi << wn )` assume that all bits for `prod_lo` and `prod_hi` arrive at the same time and that all bits of `prod` are sent to the outputs at the same time. (Don't like simplifying assumptions? The next subproblem is for you!)

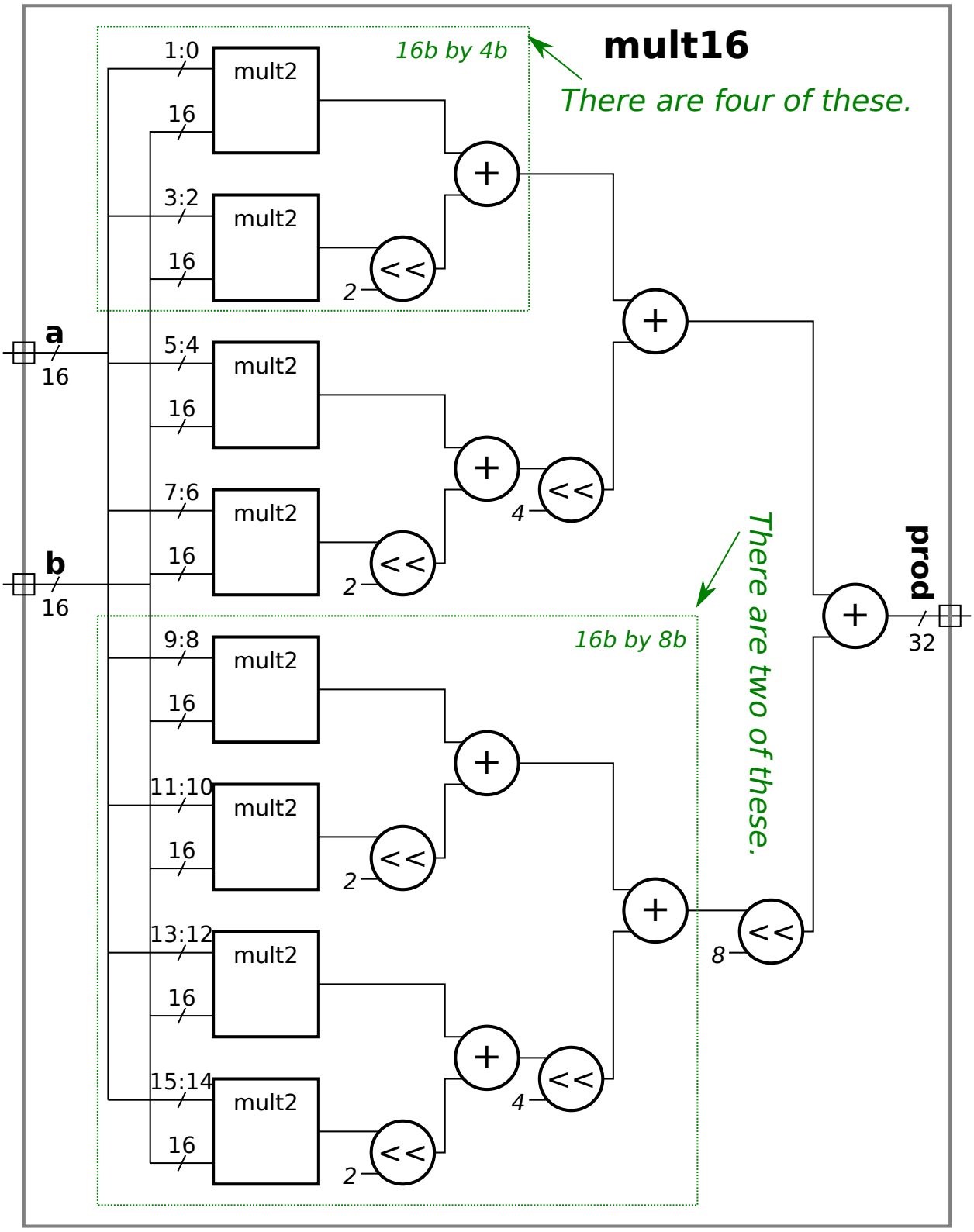
Show your answer for `w=8` and as an expression in terms of w. Don't forget to consider the entire elaboration, not just the top-level module.

(*c*) Compute the delay of the multiplier without the simplifying assumption. That is, account for the fact that the less-significant bits of `mult_tree` will be ready before the more-significant bits.

Show your answer for `w=8` and as an expression in terms of w. Don't forget to consider the entire elaboration, not just the top-level module.

Useful diagram on next page.

Use the diagram below to help work out solutions.



LSU EE 4755 Homework 4 Due: 11 November 2019

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/2019/hw04.v.html`.

Problem 0: Following instructions at `https://www.ece.lsu.edu/koppel/v/proc.html`, set up your class account (if for whatever reason you haven't done so or neeed to do it again), copy the assignment, and run the Verilog simulator and synthesis program on the unmodified homework file, `hw04.v`. Do this early enough so that minor problems (*e.g.*, password doesn't work) are minor problems.

Homework Overview

Module `best_match_behavioral` has two inputs, a longer vector, `val`, and a short vector, `k`. It sets `pos` to the start of a subvector of `val` that best matches `k` and sets `err` to the number of bit positions that don't match. For example, suppose `val = 8'b11110000` and `k=4'b1100`. Then `pos` would be set to 2 and `err` to 0 because there is an exact match at position 2 in `val`. If `k=4'b1101` then there isn't an exact match for `k` in `val`, but at position 2 there is a match with one error. If `k=2'b00` then there are matches at positions 0, 1, and 2, all with zero errors.

Module `best_match_behavioral` is combinational (and was written as a behavioral module). In this assignment a sequential version will be written and analyzed.

Testbench Code

The testbench for this assignment, which can be run when visiting the file in Emacs in a properly set-up account by pressing `F9`, tests the modules. Initially, the testbench will exit because module `best_match` has not responded in sufficient time. When that happens one of the last lines of the testbench output shows that the final cycle count is the same as the cycle limit (128 below), and "CYCLE LIMIT EXCEEDED" is shown.

```
ncsim> run
Exit from clock loop at cycle 128, limit 128.  ** CYCLE LIMIT EXCEEDED **
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit

Compilation finished at Mon Nov  4 17:56:24
```

To get rid of this message `best_match` must handshake correctly, see Problem 1. If `best_match` responds in time, the testbench will check to see if `pos` is in the right range. The output below shows errors when `pos` is out of range: `Error in best_match, test #     3, pos out of range: 0xff`

```
Error in best_match, test #     4, pos out of range: 0xff
Done with best_match_behavioral tests,           0 errors found.
Done with best_match tests,        1000 errors found.
Exit from clock loop at cycle 59001, limit 59069.
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
```

The output `err` is supposed to be the number of non-matching bits at `pos`. If not, the testbench shows output like:

```
Error in best_match, test #     2, err wrong 1 != 3 (correct) pos   2  84 ^ 01
Error in best_match, test #     3, err wrong 1 != 2 (correct) pos  13  1f ^ 3d
Error in best_match, test #     4, err wrong 1 != 2 (correct) pos   4  78 ^ f9
Done with best_match_behavioral tests,           0 errors found.
```

```
Done with best_match tests,          972 errors found.
Exit from clock loop at cycle 59001, limit 59069.
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
```

For test # 4, the testbench reports that `err` was 1 but should have been 2. The line also shows that `pos` was set to 4, and that `val` at that position was `78` (in hexadecimal) and that `k=f9`.

The testbench also checks whether the `err` returned is the minimum error for that value of `val` and `k`.

The testbench prints the details of the first few errors it finds. A grand total is printed at the end, see the transcript above.

Use Simvision to debug your modules. Feel free to modify the testbench so that it presents inputs that facilitate debugging.

Synthesis

The synthesis script, `syn.tcl`, will synthesize `best_match_behavioral` (for reference) and `best_match` (your solution). Each module will be synthesized at three widths, and with two delay targets, an easy 90 ns and a un-achievable 0.1 ns. If a module doesn't synthesize −.001 s is shown for its delay. The script is run using the shell command `genus -files syn.tcl`, which invokes Cadence Genus. If you would like to synthesize additional modules or sizes edit `syn.tcl` near the bottom.

The synthesis script shows area (cost), delay, and the delay target in a neat table. Additional output of the synthesis program is written to file `spew-file.log`.

Problem 1: Complete module `best_match` so that it computes the best match sequentially as described below. In addition to `val` and `k`, the module has 1-bit inputs `start` and `clk` and 1-bit output `ready`.

Handshaking works as follows: When `start=1` at a positive edge the module should set `ready` to zero. It should then start scanning for the best match, checking one shifted position per cycle. The maximum number of cycles needed should be `wv-wk` plus one or two more needed for handshaking. (The testbench will wait `2*wv` cycles before giving up.) The module should set `err` and `pos` to their correct values and `ready` to 1.

The inputs, `val` and `k` will be held steady at least until `ready` is set to 1.

The module must use the `pop` (population) module (in `hw04.v`) to compute possible values for `err`. That is, don't use something like the `b` loop in `best_match_behavioral` to accumulate the sum `e`. Instead compute the XOR of the appropriate bit range and provide that to the `pop` module as an input.

For maximum credit avoid the use of large (such as `wv`-input) multiplexors in your design, or the use of a non-constant shifter.

The module must be synthesizable and correct.

The behavioral best match module is shown below for reference.

```
module best_match_behavioral
  #( int wv = 32, int wk = 10, int wvb = $clog2(wv), int wkb = $clog2(wk+1) )
   ( output logic [wvb:1] pos,  // Position of best match.
     output logic [wkb:1] err,  // Number of non-matching bits.
     input uwire [wv-1:0] val, input uwire [wk-1:0] k );

   always_comb begin

      automatic int best_err = wk + 1;
      automatic int best_pos = -1;
```

```
      for ( int p=0; p<=wv-wk; p++ ) begin
         automatic int e = 0;
         for ( int b=0; b<wk; b++ ) e += k[b] !== val[p+b];
         if ( e < best_err ) begin
            best_err = e;
            best_pos = p;
         end
      end
      err = best_err;
      pos = best_pos;

   end

endmodule
```

Problem 2: Run the synthesis program and indicate how your module compares to the behavioral module.

(*a*) Compare the amount of time needed for your module compared to the behavioral one. The answer to this question requires some manipulation of the values in the `Delay Actual` column. Indicate which results are expected, and which are not expected, and explain why.

(*b*) Compare the area of your design to the behavioral one. Indicate which results are expected, and which are not expected, and explain why.

LSU EE 4755 Homework 5 Due: 20 November 2019

Problem 1: Solve 2018 Final Exam Problem 3, in which the inferred hardware for a `misc` module is to be found (a) and the state of the event queue over time simulating `misc` (b) is to be found.

Problem 2: Appearing below is a solution to Homework 4 Problem 1. Show the hardware that will be inferred for this module after some optimization. Show the `pop` module as a box.

- Clearly show all input and output ports.
- Please don't get parameters and ports confused.

```
module best_match
  #( int wv = 32, int wk = 10,  int wvb = $clog2(wv),      int wkv = $clog2(wk+1) )
   ( output logic [wvb:1] pos,  output logic [wkv:1] err,  output logic ready,
     input uwire [wv-1:0] val,  input uwire [wk-1:0] k,    input uwire start, clk );

   logic [wvb-1:0] curr_pos;
   logic [wv-1:0] sh_val;
   uwire [wkv-1:0] e;
   pop #(wk,wkv) p( e, k ^ sh_val[wk-1:0] );

   always_ff @( posedge clk )

     if ( start == 1 ) begin

        ready = 0;
        curr_pos = 0;
        sh_val = val;
        err = ~0;

     end else if ( !ready ) begin

        if ( e < err ) begin  err = e;  pos = curr_pos;  end
        ready = curr_pos == wv - wk;
        curr_pos++;
        sh_val >>= 1;

     end

endmodule
```

LSU EE 4755 Homework 6 Due: 4 December 2019

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/2019/hw06.v.html`.

Problem 0: Following instructions at `https://www.ece.lsu.edu/koppel/v/proc.html`, set up your class account (if for whatever reason you haven't done so or need to do it again), copy the assignment, and run the Verilog simulator and synthesis program on the unmodified homework file, `hw06.v`. Do this early enough so that minor problems (*e.g.*, password doesn't work) are minor problems.

Homework Overview

Module `add_accum` keeps a running total of values appearing at its inputs. A 1-bit input `ai_valid` indicates whether the value on w-bit input `ai` should be added to the total (`ai_valid==1`) or ignored (`ai_valid==0`). These signals should be examined on the positive edge of input `clk`. The module places a running sum of these values on output `sum`. The sum should be reset to 0 when input `reset` is 0 at a positive edge. The Verilog below implements the behavior described so far.

```
module add_accum
  #( int w = 20, n_stages = 3 )
   ( output logic [w-1:0] sum,
     output logic sum_valid,
     input uwire [w-1:0] ai,
     input uwire ai_valid, reset, clk );

   always_ff @ ( posedge clk )
     if ( reset ) sum = 0; else if ( ai_valid ) sum += ai;

   always_comb sum_valid = 1;

endmodule
```

A student at this point might wonder if this is going to be a dull assignment. No, of course not! Did you notice the parameter `n_stages`? That indicates that the module shall [I understand why shall is used instead of should in some contexts, but still it sounds too bossy to me] use a pipelined adder of `n_stages` stages. The point of this assignment is to modify the module above so that it uses the provided (and pre-instantiated) pipelined adder.

There are two challenges here. The straightforward challenge of connecting the pipelined input and output ports properly. Then there's the perhaps unexpected and interesting challenge of properly updating the running sum when input values arrive even while the calculation of a sum is still in the pipeline. The module has an output `sum_valid` that should only be set to `1` when output `sum` is the correct sum of all arriving valid values since the most recent reset.

After a reset `sum` should be set to zero and `sum_valid` to `1`. When the first value arrives `sum` might change to that arriving value by the next clock cycle (no adder needed). But when the second value arrives it will be necessary to add it to the first (the current sum) and since the pipelined adder takes several cycles `sum_valid` will have to be set to zero while the adder is computing. If no other new values arrive before the adder is finished `sum` can be set to the sum and `sum_valid` should again be set to `1`. Suppose instead that while the adder is operating on the first two values, a third value arrives? Then when the adder is finished the third value will have to be added to the just-completed sum. There is no restriction on when values can arrive. They may arrive every cycle or with large gaps between arrivals. If values arrive frequently then `sum_valid` may remain

0. But if values stop arriving sum_valid should eventually be set to 1 and sum should be set to the correct sum.

Testbench Code

The testbench for this assignment, which can be run when visiting the file in Emacs in a properly set-up account by pressing F9, tests add_accum instantiated for different pipeline lengths. It will check that the output values are correct, and that they don't appear too early or too late. Initially the testbench will report that there were 0 incorrect values but that they all arrived too early. The testbench will report the first four errors of each time for each pipeline length. The error message is followed by a string describing when the module was last reset and when values have since arrived. For example:

```
At cyc 7, value ready too soon, 0, cyc. (Min cyc 8.)
 R(4)+42(5)+40(7)
```

This indicates that at cycle 7 the value arrived too soon, after 0 cycles instead of after a minimum of 8 cycles. (The first value can appear after 0 cycles since there's nothing to add.) The R(4) indicates that the most recent reset was in cycle 4. The +42(5) indicates that the value 42 was at the input to the module in cycle 5.

A tally of errors and other information is shown after each pipeline length:

```
Done with 6-stage tests, 10000 series.
 Correct, 65271; errors : 0 not done, 0 val, 45273/0 early/late.
For 6 stages average latency 0.15 cycles.
```

The number after correct was the number of correct values found. To the left of "not done" is the number of tests skipped due to unresponsiveness. The number to the left of val is the number of incorrect results. The numbers to the left of early/late indicate the number of values appearing too early (45273 in the example above) or too late (0 in the example).

The testbench enforces a minimum time for all but the first value after a reset. The minimum time, n_stages, is assigned to parameter lat_min_empty in module testbench_n. The testbench enforces two maximum times. If the module is asserting sum_valid and a new value arrives, the updated sum should appear within lat_limit_empty = n_stages + 2 cycles. (That's also a testbench parameter.) If sum_valid is 0 and a new value arrives the testbench will patiently wait lat_limit_full = 2 + (1+$clog2(n_stages)) * (n_stages + 1) cycles. These testbench parameters can be changed to help with debugging, but they should be set back. The ta-bot will test the code using a different copy of the testbench module.

Following the error tally an average latency is shown, in this case less than 1 cycle. A low number is good so long as the pipelined adder is being used (which it isn't in the example above).

The following is output if the problem is solved correctly:

```
Starting tests for 2-stage pipeline.
Done with 2-stage tests, 10000 series.
 Correct, 35763; errors : 0 not done, 0 val, 0/0 early/late.
For 2 stages average latency 3.26 cycles.
Starting tests for 3-stage pipeline.
Done with 3-stage tests, 10000 series.
 Correct, 32338; errors : 0 not done, 0 val, 0/0 early/late.
For 3 stages average latency 4.64 cycles.
Starting tests for 5-stage pipeline.
Done with 5-stage tests, 10000 series.
 Correct, 28774; errors : 0 not done, 0 val, 0/0 early/late.
For 5 stages average latency 7.77 cycles.
Starting tests for 6-stage pipeline.
```

```
Done with 6-stage tests, 10000 series.
 Correct, 27737; errors : 0 not done, 0 val, 0/0 early/late.
For 6 stages average latency 9.48 cycles.
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
Total number of errors: 0
```

Use Simvision to debug your modules. Finding errors in sequential code without a debugger is time consuming and tedious. Feel free to modify the testbench so that it presents inputs that facilitate debugging.

Synthesis

The synthesis script, `syn.tcl`, will synthesize `add_pipe` (for reference) and `add_accum`. Each module will be synthesized at several pipeline depths, and with two delay targets, a delay-is-nothing-to-worry-about 10 ns and an unachievable 0.1 ns. If a module doesn't synthesize −.001 s is shown for its delay. The script is run using the shell command `genus -files syn.tcl`, which invokes Cadence Genus. If you would like to synthesize additional modules or sizes edit `syn.tcl` near the bottom.

The synthesis script shows area (cost), delay, and the delay target in a neat table. Additional output of the synthesis program is written to file `spew-file.log`.

Problem 1: Modify module `add_accum` so that it keeps an accumulated sum (see the intro above) using a pipelined adder. The module must be synthesizable. A pipelined adder has been instantiated and some starter solution code has been included:

```
module add_accum
  #( int w = 20, n_stages = 3 )
   ( output logic [w-1:0] sum,  output logic sum_valid,
     input uwire [w-1:0] ai,  input uwire ai_valid, reset, clk );

   always_ff @ ( posedge clk )
     if ( reset ) sum = 0; else if ( ai_valid ) sum += ai;

   always_comb sum_valid = 1;

   /// The code above must be removed and the pipelined adder, add_p0, used instead.

   uwire [w-1:0] aout;
   uwire [w-1:0] a0 = ai;  // May need other connections.
   uwire [w-1:0] a1;

   add_pipe #(w,n_stages) add_p0(aout,a0,a1,clk);

   logic [n_stages:0] st_occ; // Indicate which stage of add_p0 is occupied.

   uwire aout_valid = st_occ[n_stages-1];

   always_ff @( posedge clk ) if ( reset ) begin
      st_occ <= 0;
   end else begin

      // Keep track of which stage of add_p0 is occupied.
      st_occ[0] <= ai_valid;  // Lets initially assume all values enter pipe.
```

```
      // Advance other occupied signals.
      for ( int i=1; i<=n_stages; i++ ) st_occ[i] <= st_occ[i-1];
   end
endmodule
```

The module above correctly computes the accumulated sum, however it does not use the pipelined adder. The pipelined adder has been instantiated and one input has been connected (though it may need to be connected to additional items).

Beneath the pipelined adder is code needed to keep track of which stages of the adder have values. Bit `st_occ[i]` is 1 if stage `i` of the adder has a valid value. Stage 0 is initialized with the module input's valid signal. Values are advanced one position per cycle. Net `aout_valid` is 1 if the adder output is valid, which will be true `n_stages` cycles after `ai_valid` is 1.

As described in the introduction, this problem would be easy if new values arrived at least `n_stages` cycles apart, because in that case the accumulated sum and the new value could be placed in the adder without worry. But a new value can arrive while the adder is busy with one or more computations, so the new value must be buffered until there is something to add it to, either a second new value or something emerging from the pipeline.

See the checkbox items in the Verilog code for additional items to look for. A diagram like the one below might help in solving this problem.

```
Cycle      0  1  2  3  4  5  6
ai_valid   1        1

a0         ai       ai
a1         sum      sum

aout_valid 0     1  0     1
sum              =ao   =ai
Cycle      0  1  2  3  4  5  6
```

7 Fall 2018

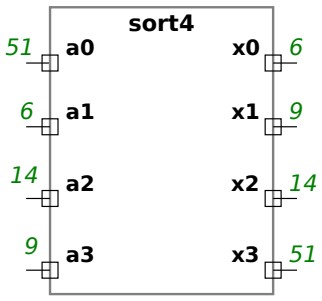
LSU EE 4755 Homework 1 Due: 5 September 2018

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/2018/hw01.v.html`.

Problem 0: Following instructions at `https://www.ece.lsu.edu/koppel/v/proc.html`, set up your class account, copy the assignment, and run the Verilog simulator and synthesis program on the unmodified homework file, `hw01.v`. Do this early enough so that minor problems (*e.g.*, password doesn't work) are minor problems.

Homework Overview

An n-input *sorting network* is a combinational circuit with n inputs and n outputs. The n values at the inputs appear at the outputs in sorted order. The illustration below shows a four-input sorting network with example values shown in green.



File `hw01.v` contains correctly functioning 2-input and 3-input sorting networks, `sort2_is`, and `sort3`. Modules `sort2` and `sort4` are empty and are to be completed for this assignment as described in the problems. File `hw01.v` contains several other modules for use in solutions, and a testbench.

Testbench Code

The testbench for this assignment, which can be run when visiting the file in Emacs in a properly set-up account by pressing F9, tests four modules: `sort2_is`, `sort2`, `sort3`, and `sort4`. Modules `sort2_is` and `sort3` should pass, the others await your solution. A sample of the end of the testbench output appears below:

```
Mod sort2, sort  2 index  0, wrong elt   z !=   0 (correct)
Tests for sort2 done, errors in         100 of         100 sorts.
Tests for sort2_is done, errors in          0 of         100 sorts.
Tests for sort3 done, errors in           0 of         100 sorts.
Mod sort4, sort  0 index  0, wrong elt   z !=  24 (correct)
Mod sort4, sort  0 index  1, wrong elt   z !=  26 (correct)
Mod sort4, sort  0 index  2, wrong elt   z !=  64 (correct)
Mod sort4, sort  0 index  3, wrong elt   z !=  94 (correct)
Mod sort4, sort  1 index  0, wrong elt   z !=   0 (correct)
Tests for sort4 done, errors in         100 of         100 sorts.
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit

Compilation finished at Tue Aug 28 16:53:25
```

A count of the number of tests and errors is shown for four modules. The testbench shows the first five errors it finds on each module, to see more modify the testbench (search for `g_elt_err_count`). In the output above the testbench is showing that the module outputs are `z` (an unconnected wire) which of course don't match the expected outputs.

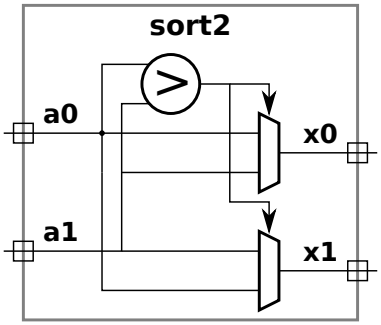
Use Simvision to debug your modules. Feel free to modify the testbench so that it presents inputs that facilitate debugging.

Synthesis

The synthesis script, `syn.tcl`, will synthesize the four modules each with two delay targets, an easy 10 ns and a un-achievable 0.1 ns. If the module doesn't synthesize −.001 s is shown for the delay. The script is run using the shell command `genus -files syn.tcl`, which invokes Cadence Genus. In past semesters Cadence RTL Compiler (`rc`) was used, which would be invoked using `rc -files syn.tcl`, **but that won't work on the 2018 homework assignments**.

The synthesis script shows area (cost), delay, and the delay target in a neat table. Additional output of the synthesis program is written to file `spew.log`. Sample synthesis script output appears below:

Problem 1: Complete module `sort2` so that it implements a 2-input sorter using a comparison unit and two 2-input multiplexors, as illustrated below. The module must pass the testbench and be synthesizable.



Use only structural code in the module (do not use `assign`, `initial`, or `always` blocks). Instantiate `mux2` for the multiplexors and `compare_le` for the comparison unit. See the check boxes in `hw01.v` near the problem for other requirements and tips.

Problem 2: Complete module `sort4` so that it implements a 4-input sorting network. Do so by instantiating `sort3` and `sort2` (or `sort2_is`) modules. As with `sort2`, use only structural code and make sure that the module passes the testbench and synthesizes.

For this assignment, implement `sort4` using one `sort3` and several `sort2` modules. Use the `sort2` modules to find the largest of the four inputs to `sort4` and connect that largest value to output `x3`. Use `sort3` to handle the remaining three values.

Implement `sort4` to minimize the critical path (measured in `sort2` or `sort2_is` modules). That is, minimize the maximum number of `sort2` (or `sort2_is`) modules traversed by any signal. The critical path for `sort3` is 3: from input `a0`, through `s0_01`, `i11`, `s1_12`, `i21`, `s2_01`, to output `x0`.

The `sort3` module uses three `sort2_is` modules. Feel free to examine `sort3` to see how modules are instantiated and interconnected.

LSU EE 4755 Homework 2 Due: 12 September 2018

Problem 1: The Verilog code below is the `sort3` module from Homework 1. Draw a diagram of the hardware as described by `sort3`, showing the `sort2` modules as boxes. Be sure to label the input and output ports with the same symbols used in the module.

```
module sort3
  #( int w = 8 )
   ( output uwire [w-1:0] x0, x1, x2,
     input uwire [w-1:0] a0, a1, a2 );

   uwire [w-1:0] i10, i11, i21;

   sort2 #(w) s0_01( i10, i11,   a0, a1 );
   sort2 #(w) s1_12( i21, x2,  i11, a2  );
   sort2 #(w) s2_01( x0, x1,  i10, i21  );

endmodule
```

Problem 2: It is possible to build an n-element sorting network using $\frac{n}{2}\lg^2 n$ two-element sorting networks in such a way that the n-element sorting network has a critical path of $\lg^2 n$. (Note: $\lg n \equiv \log_2 n$.) But this assignment is concerned with n-element sorting networks using $n(n-1)/2$ two-element sorting networks, which we will call n-element *bad sorting networks* or *bad sorters* for short.

An n-element bad sorter has inputs $a_0, a_1, \ldots, a_{n-1}$ and outputs $x_0, x_1, \ldots, x_{n-1}$. The largest value is routed to x_{n-1}.

A 2-element bad sorter is a single `sort2` module. An n-element bad sorter, $n > 2$, can be constructed using an $(n-1)$-element bad sorter and $n-1$ `sort2` modules as follows. The $n-1$ `sort2` modules are connected to the n inputs and to each other in such a way that the largest value is routed to a specific output of one of the `sort2` modules. That specific `sort2` output is connected to output x_{n-1} of the n-element sorter. The other values connect to the $(n-1)$-element bad sorter, and the $(n-1)$-element bad sorter outputs connect to outputs $x_0, x_1, \ldots, x_{n-2}$ of the n-element bad sorter that we are constructing. Note that this generalizes the solution to Homework 1 Problem 2.

The description above is recursive. At level i (the same as n above) another $i-1$ `sort2` modules are used. For a 4-element sorter we need $(4-1)+(3-1)+1 = 6$ `sort2` modules. The cost of an n-element bad sorter is found by solving the summation $\sum_{i=2}^{n} i-1$, which is $n(n-1)/2$. That's $O(n^2)$, which is how the bad sorter got its name.

It gets worse. The critical path through the bad sorter can range from bad to awful. That depends on two things: How the `sort2` modules are used to find the largest value, and how the `sort2` modules connect to the $(n-1)$-element bad sorter.

(*a*) Show the worst way that `sort2` modules can be connected to find the largest value. *Hint: the critical path should be* $n-1$ `sort2` *modules.* Provide a sketch for the general case, and an example for $n = 4$.

(*b*) Show the worst way that the `sort2` modules, as connected above, can connect to the $(n-1)$-element sorter. Provide a sketch.

(*c*) Determine the critical path for an n-element bad sorter constructed in the way described in the last two parts. *Hint: The math part should be familiar.*

(*d*) Show a much better way of connecting the `sort2` modules to find the largest value. It should be easy to show that the critical path is the lowest that is possible. Provide a sketch for $n = 8$.

The problem with the approach to building the bad sorters described in this assignment is that each level in the recursion reduces the size by 1 (that is, from n to $n - 1$), and so the critical path must be at least $O(n)$. As some students may have realized, a better approach would be to use recursion in which the n inputs were split between two $\frac{n}{2}$-element networks and then somehow combined. But how? The key insight, described by K. E. Batcher in a landmark 1968 paper, is not to try to recursively describe a sorting network, but to instead recursively describe a network that merges two already sorted sequences. The input to a 2-element merge network would be two 1-element sorted sequences. (Of course, every 1-element sequence is sorted.) Pairs of 2-element merge networks feed a 4-element merge network, and so on. This will be further described later in the semester.

LSU EE 4755 Homework 3 Due: 25 September 2018

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/2018/hw03.v.html`.

Problem 0: Following instructions at `https://www.ece.lsu.edu/koppel/v/proc.html`, set up your class account, copy the assignment, and run the Verilog simulator and synthesis program on the unmodified homework file, `hw03.v`.

Homework Overview

The sorting networks from Homework assignments 1 and 2 sorted keys only, and they only sorted unsigned integer keys. In this assignment sorter inputs will consist of keys and data, and those keys can be signed integers or floating-point values. The only module to be modified for this assignment is `sort2`.

Module `sort2` has two inputs, `a0` and `a1`, and parameters `w`, `k`, `exp`, and `sig`. Parameter `w` is the total size of each input, `k` is the size of the key, `exp` is size of the exponent (for FP keys) and `sig` is the size of the significand (for FP keys). All sizes are in bits. Each input consists of data, in bit positions `w-1:k+1`, a key type, in bit position `k`, and a key, in bit positions `k-1:0`. If bit `k` is zero the key is a **signed** integer in 2's complement representation. If bit `k` is one the key is a FP value in a format similar to IEEE 754: Bit `k-1` is the sign, bits `k-2:sig` are the exponent, and bits `sig-1:0` are the significand. For a description of these fields see the floating-point modules in the ChipWare documentation (linked to the course references page, `https://www.ece.lsu.edu/koppel/v/ref.html`, and also linked to the HTMLized assignment code, `https://www.ece.lsu.edu/koppel/v/2018/hw03.v.html`). Also see the `fp_to_val` function in the testbench code, this function converts this floating-point representation to a value.

In the unmodified file the `sort2` module compares the inputs as unsigned integers. This is wrong because the high bits of each input are data, not the keys. The `mux` connections are correct because each input should be sent to the appropriate output unmodified. The solution to the problems below involve setting `c` (the mux select signal) to the correct value.

Testbench Code

The testbench for this assignment, which can be run when visiting the file in Emacs in a properly set-up account by pressing `F9`, tests module `sort2` at two different sizes and using a mix of input types. It first tries integer-only keys (labeled `ii` in the output), then floating-point only keys (labeled `ff`), and finally integer/FP keys (labeled `if`). It reports the first five errors of each type, and for each module size reports a tally by type.

Here is a transcript showing the start of the testbench (after the compiler's own messages):

```
Starting testbench for w=32, k=16, exp=6  sig width=9...
Test ii    3, error (x0,x1): (462cf78c,7cfcf78b) != (7cfcf78b,462cf78c) correct.
           a0: data 462c, key  -2164.00000 = INT 'hf78c
           a1: data 7cfc, key  -2165.00000 = INT 'hf78b
           To re-run paste:  tests.push_back('h462cf78c); tests.push_back('h7cfcf78b);
Test ii    4, error (x0,x1): (72aed2ac,d512d2aa) != (d512d2aa,72aed2ac) correct.
           a0: data d512, key -11606.00000 = INT 'hd2aa
           a1: data 72ae, key -11604.00000 = INT 'hd2ac
           To re-run paste:  tests.push_back('hd512d2aa); tests.push_back('h72aed2ac);
```

The transcript above shows two errors, both for integer key pairs. The first line shows the actual output followed by the correct output (labeled `correct`). The number before `error` is a

test number, these start at zero and go up to `num_tests-1` (see the testbench code). The next two lines show the input values broken into data and key, including the value and representation details. The last line of each error report has text that can be put into the testbench code so that particular test can be re-run as one of the first tests.

The testbench tests the `sort2` module at two sizes. At the end of the code for each is a tally of the number of errors:

```
Done with 3000000 tests for k=16, exp=6:   499679 ff errs,   499666 if errs,   499400
ii errs,
```

In the sample above there are many errors for each type of test. Here is the output when all tests pass:

```
Starting testbench for w=32, k=16, exp=6  sig width=9...
Done with 3000000 tests for k=16, exp=6:   0 ff errs,   0 if errs,   0 ii errs,
Starting testbench for w=24, k=14, exp=5  sig width=8...
Done with 3000000 tests for k=14, exp=5:   0 ff errs,   0 if errs,   0 ii errs,

All done.
```

Debugging

To debug your code identify an error that looks easy to figure out and copy the text to the right of `paste:` into the `testbench_size` module near the comment `Add tests below`. Also change the value of `num_tests` to a small number, say 3. (Don't forget to change it back!) Verify that the code fails on test 0 (or some other small number). Next, run SimVision: `irun -gui hw03.v`. Locate your module (it will be under `t1` or `t2`) and copy symbols from `s2` to the waveform viewer. See the SimVision instructions on the `https://www.ece.lsu.edu/koppel/v/proc.html` page.

Synthesis

The synthesis script, `syn.tcl`, will synthesize `sort2` with two delay targets, an easy 10 ns and a unachievable 0.1 ns. If the module doesn't synthesize −.001 s is shown for the delay. The script is run using the shell command `genus -files syn.tcl`, which invokes Cadence Genus. In past semesters Cadence RTL Compiler (`rc`) was used, which would be invoked using `rc -files syn.tcl`, **but that won't work on the 2018 homework assignments**.

The synthesis script shows area (cost), delay, and the delay target in a neat table. Additional output of the synthesis program is written to file `spew.log`. Sample synthesis script output appears below:

Problem 1: Complete module `sort2` so that it correctly sorts inputs with **signed** integer keys. Avoid unnecessarily costly or slow designs.

Problem 2: Complete module `sort2` so that it also correctly sorts inputs with floating-point keys. Instantiate at least one ChipWare module, it's okay to use more. When adding ChipWare modules be sure to put in an `include` directive at the end of the file. Avoid unnecessarily costly or slow designs.

Problem 3: Complete module `sort2` so that it also correctly sorts inputs when one key is a signed integer and the other is floating point. Avoid unnecessarily costly or slow designs. Try to avoid solutions that use a larger significand than is specified by the parameters or other brute-force approaches.

LSU EE 4755 Homework 4 Due: 3 October 2018

Problem 1: Solve 2017 Final Exam Problem 3, in which the cost and delay of two alternative designs are to be compared.

LSU EE 4755 Homework 5 Due: 12 October 2018

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/2018/hw05.v.html`.

Problem 0: Following instructions at `https://www.ece.lsu.edu/koppel/v/proc.html`, set up your class account, copy the assignment, and run the Verilog simulator and synthesis program on the unmodified homework file, `hw05.v`.

Homework Overview

The sorting networks used in past assignments were not very efficient, they were the rough hardware equivalent of bubble sorts. In this assignment much better sorters will be implemented, sorting networks based on Batcher's odd/even merge design.

Testbench Code

The testbench for this assignment, which can be run when visiting the file in Emacs in a properly set-up account by pressing F9, tests module `batcher_sort` and `batcher_merge` at several different sizes.

Here is a transcript showing the output of the testbench (after the compiler's own messages):

```
ncsim> run
Starting testbench.
Mod batcher_merge, n=2, sort  1 idx  0, wrong elt  18 !=   7 (correct)
Mod batcher_merge, n=2, sort  1 idx  1, wrong elt   7 !=  18 (correct)
Mod batcher_merge, n=2, sort  4 idx  0, wrong elt 216 != 120 (correct)
Mod batcher_merge, n=2, sort  4 idx  1, wrong elt 120 != 216 (correct)
Mod batcher_merge, n=2, sort  7 idx  0, wrong elt 150 !=  12 (correct)
Tests for batcher_merge (idx 1)  n=2 done, errors in 3 of 10 sorts.
Tests for batcher_merge (idx 2)  n=4 done, errors in 6 of 10 sorts.
Tests for batcher_merge (idx 3)  n=8 done, errors in 10 of 10 sorts.
Tests for batcher_merge (idx 4)  n=16 done, errors in 10 of 10 sorts.
Tests for batcher_merge (idx 5)  n=32 done, errors in 10 of 10 sorts.
Tests for batcher_sort (idx 7)  n=2 done, errors in 2 of 10 sorts.
Tests for batcher_sort (idx 8)  n=4 done, errors in 10 of 10 sorts.
Tests for batcher_sort (idx 9)  n=8 done, errors in 9 of 10 sorts.
Tests for batcher_sort (idx 10)  n=16 done, errors in 10 of 10 sorts.
Tests for batcher_sort (idx 11)  n=32 done, errors in 10 of 10 sorts.
Done with all tests, errors on 10 sorters.
```

The transcript shows the first five errors in detail, this is on lines starting with `Mod`. A tally of the total number of errors for a particular module is shown on a line starting `Tests for`.

Here is the output when the assignment is correctly solved:

```
ncsim> run
Starting testbench.
Tests for Batcher Merge (idx 1)  n=2 done, errors in 0 of 10 sorts.
Tests for Batcher Merge (idx 2)  n=4 done, errors in 0 of 10 sorts.
Tests for Batcher Merge (idx 3)  n=8 done, errors in 0 of 10 sorts.
Tests for Batcher Merge (idx 4)  n=16 done, errors in 0 of 10 sorts.
Tests for Batcher Merge (idx 5)  n=32 done, errors in 0 of 10 sorts.
Tests for Batcher Sort (idx 7)  n=2 done, errors in 0 of 10 sorts.
```

```
Tests for Batcher Sort (idx 8)  n=4 done, errors in 0 of 10 sorts.
Tests for Batcher Sort (idx 9)  n=8 done, errors in 0 of 10 sorts.
Tests for Batcher Sort (idx 10)  n=16 done, errors in 0 of 10 sorts.
Tests for Batcher Sort (idx 11)  n=32 done, errors in 0 of 10 sorts.
Done with all tests, errors on 0 sorters.
```

Debugging

To debug your code run SimVision: `irun -gui hw05.v`. Locate your module and copy symbols to the waveform viewer. See the SimVision instructions on the `https://www.ece.lsu.edu/koppel/v/proc.html` page.

Synthesis

The synthesis script, `syn.tcl`, will synthesize `sort2` with two delay targets, an easy 10 ns and a unachievable 0.1 ns. If the module doesn't synthesize −.001 s is shown for the delay. The script is run using the shell command `genus -files syn.tcl`, which invokes Cadence Genus. In past semesters Cadence RTL Compiler (`rc`) was used, which would be invoked using `rc -files syn.tcl`, **but that won't work on the 2018 homework assignments**.

The synthesis script shows area (cost), delay, and the delay target in a neat table. Additional output of the synthesis program is written to file `spew.log`.

Problem 1: Complete module `batcher_sort` so that it implements a sorter as described below. The module has one input, an n-element array `a`, and one output, an n-element array `x`. Above some minimum value of `n` it should instantiate two copies of itself, each copy should sort half the the input array, `a`. A `behav_merge` module should be instantiated to merge the output of the two recursive implementations.

The `behav_merge` module, which is already written, has two inputs, `a` and `b`, each an n-element array, and one output, `x`, a $2n$-element array, where n is the value of the first parameter. Output `x` contains the elements of `a` and `b` in sorted order.

Once Problem 2 is solved correctly replace `behav_merge` with `batcher_merge`.

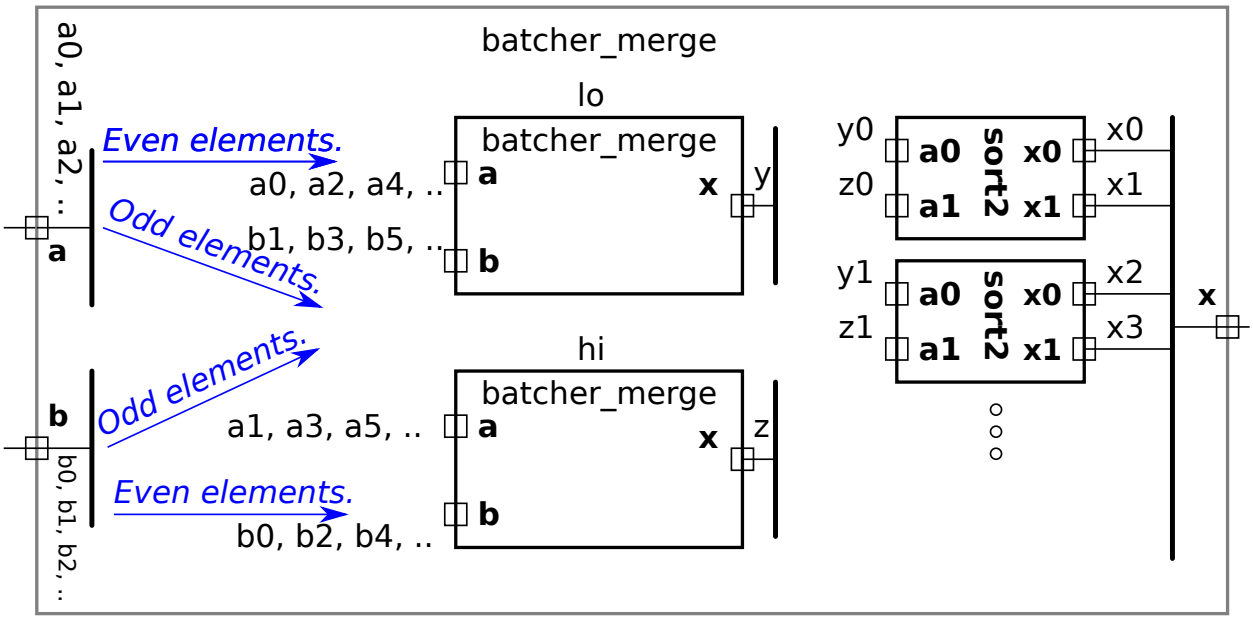
The module must use structural code, be synthesizable, be reasonably efficient, clearly written, and of course pass the testbench. See other conditions on the solution and tips in the Verilog file.

The solution to this problem is straightforward and will be in the form of other tree-structured designs shown in class.

Warning: Do not search for a solution to this problem. Exam questions will be written under the assumption that each student has solved all homework problems.

Problem 2: Complete module `batcher_merge` so that it recursively implements a Batcher odd/even merge module in which the number of elements of each input list is a power of 2. Use `sort2` instantiations to combine the output of the recursively instantiated modules. Use either structural or behavioral code to separate each input sequence into odd and even parts.

The `batcher_merge` module should recursively instantiate two copies of itself, call them lo and hi. Input `a` of the lo module should connect to the even-numbered `a` elements of the enclosing module, input `b` of lo connects to odd-numbered `b` elements of the enclosing module. For the hi module switch odd and even. See the illustration below. The illustration also shows how the outputs should connect.



Warning: Do not search for a solution to this problem. Exam questions will be written under the assumption that each student has solved all homework problems.

The module must be synthesizable, reasonably efficient, clearly written, and of course pass the testbench.

Do not compare the cost and performance reported by genus for your module, `batcher_merge`, to those for `behav_merge`. That's because genus does not correctly infer hardware for `behav_merge`.

LSU EE 4755 Homework 6 Due: 10 October 2018

Problem 1: Use the simple model to compute the cost and delay (critical path length) of the inferred hardware for module `behav_merge` from Homework 5. This module has two inputs, `a` and `b`, each of which is an n-element sorted sequence of w-bit unsigned integer values. Output `x` is a $2n$-element array of w-bit quantities. The module assigns elements of `a` and `b` to `x` so that `x` itself is a sorted sequence of the elements from `a` and `b`.

Show the cost and delay of `behav_merge` in terms of n and w. The Homework 5 module appears below. Use the tree implementation of multiplexors for cost and delay. (See the simple model notes.) Make reasonable optimizations, such as using the same multiplexor for `a[ia]` and `a[ia++]`. Avoid tedious optimizations such as varying the number of bits in `ia` and `ib`.

```
module behav_merge
  #( int n = 4, int w = 8 )
   ( output logic [w-1:0] x[2*n], input uwire [w-1:0] a[n], b[n] );

   logic [$clog2(n+1)-1:0] ia, ib;
   always_comb begin
      ia = 0; ib = 0;
      for ( int i = 0;  i < 2*n;  i++ )
        x[i] = ib == n || ia < n && a[ia] <= b[ib] ? a[ia++] : b[ib++];
   end
endmodule
```

Problem 2: As was probably mentioned, a proper n-element Batcher odd/even merge module is constructed from $\frac{n}{2}\lceil \lg n \rceil$ `sort2` modules, and the critical path length through a merge module is $\lceil \lg n \rceil$ `sort2` delays.

If the previous problem was solved correctly then the cost and critical path length of `behav_merge` should be much larger than a Batcher merge. But the behavioral code in `behav_merge` has a run time of $O(2n)$ running as an ordinary program, and consumes $O(2n)$ memory, both of which are optimal for an algorithm that must operate on all of $2n$ items. In fact, recursively applied code based on `behav_merge` can sort a sequence in $O(n \lg n)$ time, which is the best one can normally get in many cases.

What is it about the hardware realization of `behav_merge` that makes it so much less efficient than the software realization? Your answer should consider how much hardware is being used at each moment in time.

LSU EE 4755 Homework 7 Due: 16 November 2018

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/2018/hw07.v.html`.

Problem 0: Following instructions at `https://www.ece.lsu.edu/koppel/v/proc.html`, set up your class account, copy the assignment, and run the Verilog simulator and synthesis program on the unmodified homework file, `hw07.v`.

Homework Overview

Modules `mult_seq_ds_prob_1` and `mult_seq_d_prob_2` have similar sets of ports as the fast pipelined multiplier from 2017 Homework 7, but the code within this semester's modules implements a sequential rather than a pipelined multiplier. In this assignment these modules will be modified so they use handshaking to start and announce the availability of a product, and in `mult_seq_d_prob_2`, so that the latency (number of clock cycles) needed to compute a product depends upon the number and placement of zeros in the multiplicand. Unlike 2017 Homework 7, the solution to this problem **will not be pipelined**.

Initially the testbench should report errors for both modules, identified as `Prob 1` and `Prob 2`, these errors are due to the modules ignoring the handshake signals (`in_valid` and `out_avail`). The testbench, however, should correctly synthesize both modules. The testbench will also synthesize an original copy of each module, for comparison.

A correct solution to Problem 1 will eliminate the testbench errors. A correct solution to Problem 2 will reduce the number of cycles needed to compute a correct result. A future assignment or possibly final exam questions will ask about the difference in performance between the Problem 1 and Problem 2 modules.

The testbench reports details of the first few errors encountered on each modules, and then a summary. If you would like to test your module on a specific multiplier/multiplicand pair search for `values to try out` and add those to the beginning of the list assigned to `tests`. The modules are instantiated with names `prob1_m1`, `prob1_m2`, etc. Look for these when using debugging tools such as SimVision.

The synthesis script for this assignment can be run with the command `genus -files syn.tcl`. It synthesizes modules `mult_seq_ds_prob_1` and `mult_seq_d_prob_2`, as well as unmodified copies of these modules, `mult_seq_ds_prob_1_orig` and `mult_seq_d_prob_2_orig`. Each is synthesized at two sizes and three different values of m. The synthesis script assumes at latency of $\lceil w/m \rceil$ for these modules, which is an overestimate for Problem 2.

Problem 1: Module `mult_seq_ds_prob_1` has two parameters, `w` and `m`, four input ports, `clk`, `in_valid`, `plier`, and `cand`, and two output ports, `prod`, and `out_avail`.

The unmodified module will set `prod` to the $2w$-bit product of w-bit inputs `plier` and `cand`, which hold unsigned integers. It computes the product using m-bit partial products, similar to the method used by `mult_seq_dm` but using the streamlined code in `mult_seq_stream`. In `mult_seq_ds_prob_1` the product will be available with a latency of between $\lceil w/m \rceil + 1$ and $2\lceil w/m \rceil - 1$ cycles. The latency will be $\lceil w/m \rceil + 1$ when the multiplier and multiplicand arrive when `iter` is reset to zero, but if they arrive one cycle later the latency will be $2\lceil w/m \rceil - 1$. If that higher latency bothers you then this is your problem. (Even those that don't care need to solve this problem.)

The reason for this variation in latency in `mult_seq_dm` and friends is that those modules have no way of knowing when a new multiplier/multiplicand pair has arrived (other than continually comparing them to prior values which would require extra hardware). As the alert student

suspects, input `in_valid` in `mult_seq_ds_prob_1` is used to indicate the arrival of a new pair. Modify `mult_seq_ds_prob_1` so that it starts a new multiplication at the positive edge of `clk` when `in_valid` is `1`, even if there's a multiplication in progress. When a new multiplication starts set `out_avail` to `0`, and set it back to `1` when `prod` holds the correct product.

When this problem is correctly solved the testbench should not show errors on this module. The testbench instantiates the module for three sizes of m, and it has `Prob 1` in the name. See the checkboxes in `hw07.v` for additional requirements and tips. Don't forget synthesizability as well as clear and efficient code.

Problem 2: The unmodified module `mult_seq_d_prob_2` computes a product in at best $\lceil w/m \rceil + 1$ cycles. For some multiplicands the value of `cand_2d[iter]` (see the code) will be zero for certain values of `iter`. An extreme case is when the multiplicand is zero, but there are many other situations where `cand_2d[iter]` will be zero. Currently `iter` is incremented by 1 each clock cycle. Modify `mult_seq_d_fast` so that `iter` is incremented so that it points to the next non-zero value in `cand_2d`, or to $\lceil w/m \rceil$ if there are no more non-zero values. Doing so will reduce the number of clock cycles needed to compute a product. This should be reflected in the `Avg Cyc` shown for the `Prob 2` module by the testbench.

Use the synthesis script `syn.tcl` to find the clock period of the module. The latency shown by the synthesis script assumes w cycles per multiply. To find the actual latency of your module multiply the clock period reported by the synthesis script with the average cycles reported by the testbench.

The goal is to reduce the average number of cycles, as reported by the testbench while also keeping clock period low (as reported by the synthesis script) so that the average latency, measured in seconds (or some fraction) is lower.

LSU EE 4755 Homework 8 Due: 27 November 2018

Problem 1: Appearing below is the output of the simulator and synthesis script, showing data for the Homework 7 solution modules. Modules are simulated and synthesized for $w = 32$.

```
Module Name                              Area  Period  Period
                                               Target  Actual
mult_seq_ds_prob_1_w32_m1              157813    1000   14926
mult_seq_ds_prob_1_w32_m2              185493    1000   15431
mult_seq_ds_prob_1_w32_m4              242568    1000   16296

mult_seq_d_prob_2_w32_m1               288580    1000   31944
mult_seq_d_prob_2_w32_m2               301203    1000   32204
mult_seq_d_prob_2_w32_m4               329226    1000   32192

For Prob 1 Deg 1       ran  400 tests,    0/   0/   0 errors found. Avg cyc 33.0
For Prob 1 Deg 2       ran  400 tests,    0/   0/   0 errors found. Avg cyc 17.0
For Prob 1 Deg 4       ran  400 tests,    0/   0/   0 errors found. Avg cyc 9.0

For Prob 2 Deg 1       ran  400 tests,    0/   0/   0 errors found. Avg cyc 9.5
For Prob 2 Deg 2       ran  400 tests,    0/   0/   0 errors found. Avg cyc 7.3
For Prob 2 Deg 4       ran  400 tests,    0/   0/   0 errors found. Avg cyc 5.0
Modules instantiated with w = 32.
```

The Problem 1 modules are based on the streamlined multiplier and so are faster. But the Problem 2 modules skip zeros. Based on the data above, indicate the ways, if any, that the Problem 2 modules are better than the Problem 1 modules. Explain using the numbers above.

There are more problems on the next pages.

Problem 2: Appearing below is a solution to Homework 7, Problem 1, the streamlined degree-m multiplier with handshaking. The complete solution is at `https://www.ece.lsu.edu/koppel/v/2018/hw07-sol.v.html`. For this problem assume that w and m are both powers of 2.

```
module mult_seq_ds_prob_1 #( int w = 16, int m = 2 )
   ( output logic [2*w-1:0] prod,  output logic  out_avail,
     input uwire clk, in_valid,    input uwire [w-1:0] plier, cand );

   localparam int iterations = ( w + m - 1 ) / m;
   localparam int iter_lg = $clog2(iterations);

   uwire [iterations-1:0][m-1:0] cand_2d = cand;

   bit [iter_lg:0] iter;
   logic [2*w-1:0] accum;

   always_ff @( posedge clk ) begin

      if ( in_valid ) begin

         accum = cand;
         iter = 0;
         out_avail = 0;

      end else if ( !out_avail && iter == iterations ) begin

         out_avail = 1;
         prod = accum;

      end

      accum = { 0 + plier * accum[m-1:0] + accum[2*w-1:w], accum[w-1:m] };
      iter++;
   end

endmodule
```

(*a*) Show the hardware that will be inferred for this module. The Inkscape SVG format diagram of the hardware for the streamlined sequential module from the class demo notes can be used as a starting point. It is at `https://www.ece.lsu.edu/koppel/v/2018/ill-mul-seq-str.svg`.

(*b*) Compute the cost and delays for this module using the simple model. Show these in terms of w and m. Clearly show the critical path on your diagram.

There is a problem on the next page.

Problem 3: Appearing below is a solution to Homework 7, Problem 2, the streamlined degree-m multiplier with handshaking. The complete solution is at `https://www.ece.lsu.edu/koppel/v/2018/hw07-sol.v.html`. For this problem assume that w and m are both powers of 2.

```
module mult_seq_d_prob_2 #( int w = 16, int m = 2 )
   ( output logic [2*w-1:0] prod,   output logic out_avail,
     input uwire clk, in_valid,     input uwire [w-1:0] plier, cand );

   localparam int iterations = ( w + m - 1 ) / m;
   localparam int iter_lg = $clog2(iterations);

   uwire [iterations-1:0][m-1:0] cand_2d = cand;

   bit [iter_lg-1:0] iter;
   logic [2*w-1:0] accum;

   always_ff @( posedge clk ) begin

      logic [iter_lg-1:0] next_iter;

      if ( in_valid ) begin
         iter = 0;
         accum = 0;
         out_avail = 0;
      end else if ( !out_avail && iter == 0 ) begin
         prod = accum;
         out_avail = 1;
      end

      accum += plier * cand_2d[iter] << ( iter * m );

      next_iter = 0;
      for ( int i=iterations-1;  i>0;  i-- )
        if ( i>iter && cand_2d[i] ) next_iter = i;
      iter = next_iter;
   end

endmodule
```

(a) Show the hardware that will be inferred for this module.

(b) Compute the cost and delays for this module using the simple model. Show these in terms of w and m. Clearly show the critical path on your diagram.

8 Fall 2017

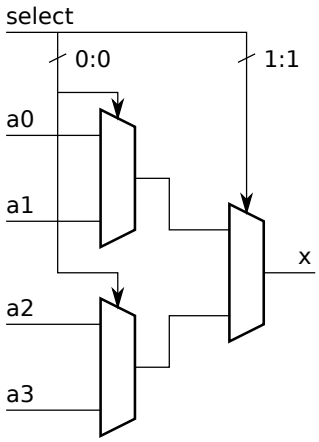
LSU EE 4755 Homework 1 Due: 8 September 2017

Start working on the solutions to the problems below on paper, but complete them using the computers in the lab. For instructions visit `https://www.ece.lsu.edu/koppel/v/proc.html`. *For the complete Verilog for this assignment without visiting the lab visit* `https://www.ece.lsu.edu/koppel/v/2017/hw01.v.html`.

Problem 1: Appearing below, and in `hw01.v`, is a Verilog description of a 2-input multiplexer, `mux2`, and a partially completed description of a 4-input mux, `mux4`, along with a diagram showing how a four-input mux can be made using three two-input multiplexers. Complete `mux4` as described in the diagram.

It is important that `mux4` instantiate three `mux2` modules. Other correct 4-input multiplexer implementations will not receive credit. Also, don't forget to set the parameters correctly when instantiating modules.

```
module mux2
  #( int w = 16 )
   ( output uwire [w-1:0] x,
     input uwire s,
     input uwire [w-1:0] a, b );

   assign  x = s == 0 ? a : b;

endmodule
module mux4
  #( int w = 6 )
   ( output uwire [w-1:0] x,
     input uwire [1:0] s,
     input uwire [w-1:0] a[3:0] );

endmodule
```



Problem 2: Appearing below is a `mux8` module. Complete `mux8` so that it implements an 8-input multiplexer using two `mux4` modules and one `mux2` module. Notice that the data input to `mux8` is an 8-element array of w-bit quantities. To see how to extract a subrange of an array (called a *part select* in Verilog) see the `testbench` module. Solve this problem by generalizing the technique appearing in the previous problem.

Credit will only be given for `mux8` modules that *instantiate* two `mux4` modules and a `mux2` module. Yes, `assign x = a[s];` is correct and the best way to do it in other situations, but the goal here is to learn about instantiation.

```
module mux8
  #( int w = 5 )
   ( output uwire [w-1:0] x,
     input uwire [2:0] s,
     input uwire [w-1:0] a[7:0] );


endmodule
```

Appearing below is the start of the testbench code. To see the complete testbench and other modules follow `https://www.ece.lsu.edu/koppel/v/2017/hw01.v.html`.

```
module testbench();

   localparam int w = 10;
   localparam int n_in_max = 8;
   localparam int n_mut = 3;

   uwire [w-1:0] x[n_mut];
   logic [2:0]  s;
   logic [w-1:0] a[n_in_max-1:0];

   mux2 #(w) mm2(x[0], s[0], a[0], a[1]);
   mux4 #(w) mm4(x[1], s[1:0], a[3:0]);
   mux8 #(w) mm8(x[2], s[2:0], a[7:0]);

   initial begin

      automatic int n_test = 0;
      automatic int n_err = 0;
```

LSU EE 4755 Homework 2 Due: 25 September 2017

For instructions visit `https://www.ece.lsu.edu/koppel/v/proc.html`. *For the complete Verilog for this assignment without visiting the lab visit* `https://www.ece.lsu.edu/koppel/v/2017/hw02.v.html`.

Problem 1: Let $f\|\Re \rightarrow \Re$ be an approximately linear function, for example, $f(x)$ might be the height of a tree that has been growing for x years. Suppose further that we have two pairs of values for the function, $a_1 = f(x_1)$ and $a_2 = f(x_2)$ with $x_1 < x_2$. (For example, $x_1 = 1$ year and $a_1 = 1\,\text{m}$, and $x_2 = 2$ years and $a_2 = 1.5\,\text{m}$.) To compute values of x between x_1 and x_2 we can use linear interpolation $a(x) = a_1 + (x - x_1)\frac{a_2-a_1}{x_2-x_1}$. Notice that $a(x_1) = a_1$ and $a(x_2) = a_2$.

Module `interp_behav` has four 32-bit inputs `x1`, `a1`, `x2`, and `a2`. Each will hold a number in `shortreal` representation, their values indicate the endpoints of a region to linearly interpolate, as described in the previous paragraph. Input `j`, `jw` bits, is expected to be $\lfloor x - x_1 \rfloor$, where x is a value to interpolate. (For example, suppose $x_1 = 20$. For $x = x_1 = 20$, input `j` would be `0`, for $x = 25.7$, `j=5`.) Output `aj` is an 8-bit integer and is to be set to $\lfloor a(x_1 + j) \rfloor$. Though input `j` and output `aj` are integers, it's important that some of the calculation be done in floating-point to avoid rounding errors, such as the computation $\frac{a_2-a_1}{x_2-x_1}$. (Note: If you're having trouble following this, don't worry. All you really need to do is to look at what `interp_behav` is doing.) Finally, 1-bit output `valid` is set to `1` if $\lfloor x_1 \rfloor + j \leq \lfloor x_2 \rfloor$ and `0` otherwise.

```
module interp_behav
  #( int jw = 12,  amax = 255 )
   ( output logic valid,                    output logic [7:0] aj,
     input uwire [31:0] x1, a1, x2, a2,     input uwire [jw-1:0] j );

   always_comb begin

      automatic shortreal x1r = $bitstoshortreal(x1);
      automatic shortreal x2r = $bitstoshortreal(x2);
      automatic shortreal a1r = $bitstoshortreal(a1);
      automatic shortreal a2r = $bitstoshortreal(a2);

      automatic int x1i = $floor(x1r);
      automatic int x2i = $floor(x2r);
      automatic int xj = x1i + j;

      shortreal dadx, ajr;

      valid = xj <= x2i;

      dadx = ( a2r - a1r ) / ( x2r - x1r );
      ajr = a1r + j * dadx;
      aj = ajr < 0 ? 0 : ajr > amax ? amax : $floor(ajr);

   end

endmodule
```

The code in `interp_behav` computes these values using behavioral code. It is not synthesiz-

able because it uses operators to perform floating-point arithmetic. Module `interp` has the same connections as `inter_behav`, and has some starter code. Modify module `interp` so it computes the same values and is synthesizable. To do so instantiate ChipWare modules to perform floating-point operations. Module `interp` already instantiates an adder and a float-to-int converter. Find additional modules in the ChipWare documentation, which can be found on the course references page. When using a ChipWare module **put in an include directive at the end of the file**. See the end of `hw02.v` for examples.

```
module interp
  #( int jw = 12, amax = 255 )
   ( output logic valid,                    output logic [7:0] aj,
     input uwire [31:0] x1, a1, x2, a2,     input uwire [jw-1:0] j );

   uwire [jw:0] x1i, x2i;

   fp_ftoi #( jw+1 ) ftoi1(x1i, x1);
   fp_ftoi #( jw+1 ) ftoi2(x2i, x2);

   uwire [31:0] sum;
   fp_add add1(sum, x1, x2);  // An instantiation example, not otherwise useful.

   // These are obviously incorrect, but they avoid synthesis errors.
   assign      aj = {1'b0, sum[6:0]};
   assign      valid = sum[8];

endmodule
```

The testbench will test module `interp`, it should initially report lots of errors. Of course, when you are done there should be zero errors.

Follow the synthesis steps on the course procedures page to determine if `interp` is synthesizable. If the elaborate step is successful then the module is synthesizable.

Problem 2: Floating-point hardware is relatively costly. Compare the cost of FP and integer arithmetic units by synthesizing equivalent FP and integer adders and dividers. Wrap the ChipWare modules in your own modules, (such as `fp_add` in `hw02.v`) and set parameters so the FP and integer units are comparable. Then modify the synthesis script, `syn.tcl`, so that it will synthesize these modules. The modules should be added to the list assigned on the `set modules` line.

Based on the data collected above, indicate how much less you think the cost would be of an `interp` module that used integer arithmetic.

LSU EE 4755 Homework 3 Due: 4 October 2017

Problem 1: Solve 2016 EE 4755 Final Exam Problem 2, in which timings are requested for individual units, such as a BFA and more complex circuits made from individual units.

In the simple timing model 2-input AND and OR gates each have a delay of 1 unit, and NOT gates have a delay of 0 units. AND and OR gates with more than two inputs have the delay obtained with a reduction tree of 2-input gates. That is, n-input AND and OR gates have a delay of $\lceil \lg n \rceil$ units.

Problem 2: Solve 2016 EE 4755 Final Exam Problem 4, in which the cost of some circuits is to be computed.

In the simple cost model NOT gates have a cost of 0 units and n-input AND and OR gates each have a cost of $n-1$ units. The cost of other gates is the cost of the AND, OR, and NOT gates needed to implement them.

LSU EE 4755 Homework 4 Due: 1 November 2017

For instructions visit `http://www.ece.lsu.edu/koppel/v/proc.html`. *For the complete Verilog for this assignment without visiting the lab visit* `http://www.ece.lsu.edu/koppel/v/2017/hw04.v.html`.

Problem 1: A *run* of characters in a sequence is a set of consecutive characters that are the same, the length of a run is the number of times the character is repeated in the run. For example, the string `aabbbcaaaa` contains four runs: a run of length 2 for character `a`, a run of length 3 for character `b`, a run of length 1 for character `c`, and a run of length 4 for character `a`. (Note that `a` has two runs.)

Module `maxrun`, when completed, will keep track of the maximum-length run in a sequence of characters appearing at its `in_char` input. In this assignment *character* refers to a c-bit value. The testbench (including the excerpt below) shows character values as two-digit hexadecimal numbers. For example, at cycle 2 the table shows `c=8d` for `in_char`, meaning that the character value is $8d_{16} = 141_{10}$. The fact that the character can be represented using two hexadecimal digits or three decimal digits does not change the fact that it is one character.

At the positive edge of input `clk`, `maxrun` will compare the character at `in_char` to the character seen at the previous positive edge. If they are the same it will increment a *current run counter*, if the characters are different it will set the current run counter to 1. If `reset` is `1` at the clock positive edge then the current run counter should be set to 1 (which is the same as setting it to 0 and then incrementing it). A second *max run counter* is also set to zero on the `reset` signal at the positive clock edge. If the current run counter is greater than the max run counter then the max run counter is set to the current run counter, and the character appearing in that run is remembered and used used for output `mr_char`.

If input `mr` is 1, then output `len` should be set to the value of the max run counter, otherwise it should be set to the value of the current run counter. This should be done asynchronously (`len` should be updated whenever `mr` changes, not just at a positive clock edge).

For example, look at the output of the testbench below. Column `R` shows the value of the reset signal and column `in_char` shows the input character, both appearing before and "during" the positive clock edge. The remaining columns show the value of the current run counter, max run counter, and the `mr_char` output after the positive clock edge. At cycle 1 the input character, `00`, has a run of 2. At cycle 6 character `8d` has reached a run of 5, etc. The testbench shows an `ok` if the output is correct, otherwise it shows what the correct output should be.

```
Cycle R in_char current-run          max-run             mr_char
----- - -----   -------------        -------------       ----------
    0 r c=00    cr_len   1 ok        mr_len   1 ok        mr_c 00 ok
    1   c=00    cr_len   2 ok        mr_len   2 ok        mr_c 00 ok
    2   c=8d    cr_len   1 ok        mr_len   2 ok        mr_c 00 ok
    3   c=8d    cr_len   2 ok        mr_len   2 ok        mr_c 00 ok
    4   c=8d    cr_len   3 ok        mr_len   3 ok        mr_c 8d ok
    5   c=8d    cr_len   4 ok        mr_len   4 ok        mr_c 8d ok
    6   c=8d    cr_len   5 ok        mr_len   5 ok        mr_c 8d ok
    7   c=c5    cr_len   1 ok        mr_len   5 ok        mr_c 8d ok
    8   c=77    cr_len   1 ok        mr_len   5 ok        mr_c 8d ok
    9   c=f2    cr_len   1 ok        mr_len   5 ok        mr_c 8d ok
   10 r c=f2    cr_len   1 ok        mr_len   1 ok        mr_c f2 ok
   11   c=f2    cr_len   2 ok        mr_len   2 ok        mr_c f2 ok
   12   c=f2    cr_len   3 ok        mr_len   3 ok        mr_c f2 ok
```

```
13   c=f2    cr_len   4 ok        mr_len   4 ok        mr_c f2 ok
14 r c=f2    cr_len   1 ok        mr_len   1 ok        mr_c f2 ok
15   c=9d    cr_len   1 ok        mr_len   1 ok        mr_c f2 ok
16   c=0d    cr_len   1 ok        mr_len   1 ok        mr_c f2 ok
17   c=d5    cr_len   1 ok        mr_len   1 ok        mr_c f2 ok
```

Complete the `maxrun` module so that it passes the testbench and is synthesizable. Please pay attention to the parameters, which indicate the size of a character and the number of bits in the counters. Use the parameters, not their default values.

Use `simvision` for debugging, which is explained in the course procedures page.

Problem 2: Run the synthesis script, using command `rc -files syn.tcl`. If it runs correctly, a file `spew-file.log` will be created which contains a timing report for a design. On paper or in comments in the submission file, indicate where the critical path is in your design.

Provide suggestions on making it faster, or explain what you actually did for a high clock frequency.

LSU EE 4755 Homework 5 Due: 10 November 2017

For instructions visit `http://www.ece.lsu.edu/koppel/v/proc.html`. *For the complete Verilog for this assignment without visiting the lab visit* `http://www.ece.lsu.edu/koppel/v/2017/hw05.v.html`.

Problem 1: Module `lookup_char` has a w-bit input `char`, and two outputs, `found` and `idx`. The module has parameter `chars`, which is an n-element array of w-bit characters. Complete `lookup_char` so that output `found` is logic `1` iff `char` is equal to one of the elements of `chars`. Set `idx` to the index of that character. (That is, if `found` is `1` then `chars[idx] == char`.) It does not matter what `idx` is if the character is not found. The module should synthesize into combinational logic.

See the Verilog Problem 1 code for details on the parameters and ports and review the comment checkboxes at the top of the problem for additional tips.

Module `lookup_char` will be used in the next problem and the testbench will be able to test `lookup_char` even if no other parts of `nest` are finished.

Note: There is a 2016 EE 4755 homework assignment in which a module a lot like `lookup_char` had to be designed. The major difference is that in 2016 the `chars` array was a port, here it is a parameter. Feel free to look at the solutions. It should go without saying that the `chars` array should remain a parameter in this assignment.

Problem 2: Module `nest`, when completed, will monitor a sequence of characters that includes bracketing characters (such as parentheses) and indicate whether these bracketing characters are properly nested. For example, sequence "a()[d()e[f]]" is properly nested but "a(]" is not.

The module has input parameters `char_open` and `char_close`, each of these is an n-element array of w-bit characters listing characters that are to be treated as opening and closing bracketing characters. See the Verilog code for details. The module has three inputs, `clk`, `reset`, and `in_char`. The module has five outputs, `level`, `awaiting`, `is_open`, `is_close`, and `bad`.

Output `is_open` should be set to `1` iff `in_char` is one of the characters in `char_open`, and `is_close` should be set to `1` iff `in_char` is one of the characters in `char_close`. These outputs should be generated by instantiations of `lookup_char` (the module from the first problem). The logic for computing `is_open` and `is_close` should be combinational.

(*a*) Complete the logic for `is_open` and `is_close` as described above. The testbench checks these outputs for correctness, look for `op` and `cl` in the trace. They are correct if `er` does not appear to the right of the `0` or `1`. The module must be synthesizable.

The module has an output `level` which should operate as follows. On a positive clock edge if `reset` is `1`, `level` is set to zero. Otherwise, if `in_char` is in `char_open` then `level` should be incremented and if `in_char` is in `char_close` then `level` should be decremented. If `in_char` is in neither list then `level` is left unchanged. (`level` provides the current nesting level. A value of 0 indicates the current character is not "inside" any bracketing characters, or put another way, that we are not awaiting something like a closing parenthesis.)

The module has an output `bad` which indicates whether the sequence seen since the last reset is improperly nested or if the nesting level exceeded `d`, a module parameter. Set `bad` to 0 when `reset` is 1 (at a positive clock edge). Set `bad` to 1 if a closing character is seen when `level` is 0 or if an opening character is seen when `level` is `d`.

Also set `bad` to 1 if the wrong closing character is seen. For example, for "(]" set `bad` to 1 when the "]" is seen because a ")" was expected.

Output `awaiting` should be set to the next valid closing character. For example, if the sequence so far is "()[" `awaiting` should be set to "]".

When `bad` is `1` outputs `level` and `awaiting` can be set to any value.

Note that `bad`, `level`, and `awaiting` should be updated at the positive clock edge.

(*b*) Complete `nest` so that it works as described above. The module must be synthesizable and show no errors.

The testbench checks `nest` for correctness and at the end of a run it shows the number of errors. As of this writing it will test `nest` on 1000 different sequences, see variable `num_groups` in the testbench. It will print details on up to 2 sequences with zero errors and up to 3 sequences with at least one error. Feel free to edit the testbench to change these numbers.

Consider the following sample of testbench output:

```
ncsim> run

cyc    2  s.c  0. 0  i  op 0     cl 0     bad 0     lev  0  0     await ' )'
cyc    3  s.c  0. 1  J  op 0     cl 0     bad 0     lev  0  0     await ' )'
```

The text `cyc 2` indicates the cycle number. That can be used with SimVision or some other tool to locate the place in execution. The text `s.c 0. 1` shows the sequence number, 0, and the number of previous characters in the sequence, 1. Next shown is the character at `in_char`, `J` in cycle 3. The text `op 0 cl 0 bad 0` show the values of the `is_open`, `is_close`, and `bad` outputs that `nest` has produced. If these values are wrong then the text `er` appears to the right of the value. For example if the value at the `is_open` port were wrong the text would be `op 0 er` . Note that `bad 1` is fine but `bad 0 er` indicates that the `bad` port value is wrong. The text `lev 0 0` shows both the module `level` output (the first `0` here) and the known correct value (the second `0`). Finally, `await` shows the module output followed by the correct value. They are between quotes to make spaces and other non-printable characters obvious.

Note that when level is zero the value of await is irrelevant. Also, when `bad` is 1, the values of level and await are both irrelevant.

The example below shows the trace output when there are errors:

```
cyc   54  s.c  5. 0  L  op 0     cl 0     bad 0     lev  0  0     await ' )'
cyc   55  s.c  5. 1  (  op 1     cl 0     bad 0     lev  0  1 er  await ' )'
cyc   56  s.c  5. 2  (  op 1     cl 0     bad 0     lev  0  2 er  await ' )'
cyc   57  s.c  5. 3  q  op 0     cl 0     bad 0     lev  0  2 er  await ' )'
cyc   58  s.c  5. 4  Z  op 0     cl 0     bad 0     lev  0  2 er  await ' )'
cyc   59  s.c  5. 5  )  op 0     cl 1     bad 1 er  lev  7  1     await ' )'
cyc   60  s.c  5. 6  )  op 0     cl 1     bad 1 er  lev  6  0     await ' )'
```

At cycle 55 level should have been incremented for the "(", but it was not. Notice the `er` to the right of `lev`. Also, at cycle 59 the module set `bad` to 1 which is an error because the sequence has not violated any rules.

Problem 3: Run the synthesis script, using command `rc -files syn.tcl`. If it runs correctly, a file `spew-file.log` will be created which contains a timing report for a design. On paper or in comments in the submission file, indicate where the critical path is in your design.

Provide suggestions on making it faster, or explain what you actually did for a high clock frequency.

LSU EE 4755 Homework 6 Due: 13 November 2017

Problem 1: The solution to Homework 4, `http://www.ece.lsu.edu/koppel/v/2017/hw04-sol.v.html`, includes two modules, `maxrun` and `maxrun_opt`.

(*a*) Show the hardware inferred for `maxrun`.

(*b*) Show the hardware inferred for `maxrun_opt`.

Problem 2: Compute the critical path for the `maxrun` and `maxrun_opt` modules using the simple model. The launch points (path starts) are at module inputs and register outputs, and the capture points (path ends) are at module outputs and register inputs. Note that with these definitions the critical path does not include the register itself. Show the critical path in terms of w, the number of bits in the `len` output and c the number of bits in a character.

LSU EE 4755 Homework 7 Due: 29 November 2017

For instructions visit `http://www.ece.lsu.edu/koppel/v/proc.html`. *For the complete Verilog for this assignment without visiting the lab visit* `http://www.ece.lsu.edu/koppel/v/2017/hw07.v.html`.

Problem 1: Module `mult_pipe` is a simple pipelined multiplier which multiplies two `w`-bit operands, computing `m` partial products per stage in $\lceil w/m \rceil$ stages. The latency of this multiplier is $\lceil w/m \rceil$ cycles regardless of what is being multiplied, which in many circumstances is just fine.

In contrast `mult_fast` is designed for situations in which lower latency is beneficial. The goal is to compute the results for "easier" products in fewer cycles. For example, multiplying $abcd_{16} \times 9876_{16}$ in a 16-bit degree-4 (`m=4`) multiplier would take four cycles since all partial products are needed. But, $abcd_{16} \times 1_{16}$ requires one partial product and so the product should be available sooner.

Like the other multipliers `mult_fast` has `w`-bit inputs `plier` and `cand` and a `2w`-bit output `prod`, and a 1-bit `clk` input. But it also has a 1-bit input `in_valid` and a 1-bit output `out_avail`.

At each positive clock edge if input `in_valid` is `1` `mult_fast` should start computing the product of the input values, `plier`×`cand`. If input `in_valid` is `0` then the external hardware does not need `plier`×`cand`. Though the module can start computing `plier`×`cand` when `in_valid` is `0`, it should not set `out_avail` when the product is ready.

The outputs `out_avail` and `prod` should be set at each positive clock edge. If `out_avail` is `1` then `prod` is the product of values appearing earlier at the inputs at a time when `in_valid` was `1`. The products should appear in the same order as the inputs. For example, suppose in cycle 10 the values $8765_{16} \times 53ab_{16}$ appear at the inputs and at cycle 11 the values 1×1 appear. Even though 1×1 can be computed in one cycle it cannot appear at the outputs until after $8765_{16} \times 53ab_{16}$ appears. If it takes four cycles to compute $8765_{16} \times 53ab_{16}$ then it will appear at the outputs in cycle 14, and so the product 1×1 will not appear at the outputs until four cycles after it arrives, at cycle $11 + 4 = 15$.

A simple case is when `in_valid` is always equal to one. In that case after `w/m` cycles `out_avail` should always be set to one and the value at output `prod` is the product of inputs appearing `w/m` cycles earlier, which is how an ordinary pipelined multiplier, such as `mult_pipe` operates.

Next, consider the table below which shows inputs and possible outputs. In cycle 0 the values 1×11 arrive. Their product, 11 appears at the outputs in cycle 1. In cycle 1 values `98` and `99` appear at the inputs but since `in_valid` is `0` their product is not needed. At cycle 2 values `3` and `22` are at the inputs, the product $3 \times 22 = 66$ appears at the output in cycle 4. Note that at cycles 2 and 3 `out_avail` is `0`. The product 4×14 appears at the outputs in cycle 5.

cycle	0	1	2	3	4	5
in_valid	1	0	1	1		
plier	1	98	3	4		
cand	11	99	22	14		
out_avail	0	1	0	0	1	1
prod		11			66	56

Note that it took two cycles to compute 3×22 but one cycle to compute the other products.

(*a*) Modify `mult_fast` so that it sets `out_avail` when a product is ready. If this is completed correctly the testbench should show that there are zero errors.

(*b*) Modify `mult_fast` so that the product is ready when all of the remaining multiplicand bits are zero. That is, suppose stage i examines bits mi to $mi + m - 1$ of the multiplicand. If multiplicand bits $w - 1$ to $mi + m - 1$ are all zero then the product is finished at stage i. If this is completed

correctly then the testbench should show that the average number of cycles for the degree-2 fast multiplier is about 5.1 and for degree 4 it should be about 2.7.

- Modules must be synthesizable.
- Modules must be reasonably efficient.
- Do not assume specific parameter values.
- Use SimVision for debugging.

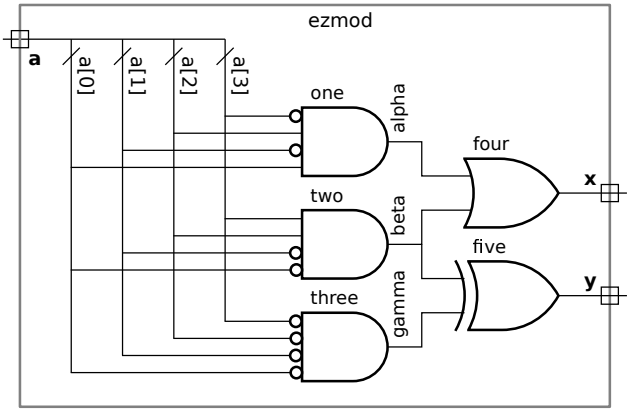
9 Fall 2016

LSU EE 4755 Homework 1 Due: 9 September 2016

The questions below can be answered without using EDA software, paper and pencil will suffice. Please turn in the solution on paper. Homework 2 will require the use of Verilog implementations. Nevertheless, runnable SystemVerilog code for this assignment can be found at `https://www.ece.lsu.edu/koppel/v/2016/hw01.v` *(plain Verilog) and* `https://www.ece.lsu.edu/koppel/v/2016/hw01.v.html` *(syntax-highlighted HTML).*

Those who are rusty about the correspondence between Verilog code and hardware might want to look at the solution to EE 3755 Fall 2013 Homework 1, at `http://www.ece.lsu.edu/ee3755/2013f/hw01_sol.pdf`.

Problem 1: Show a Verilog explicit structural description of the module illustrated below. In this assignment it is okay to use primitives (`and, not,`...), but don't get in the habit of using them.



- Base the names of ports, wires, and instances on labels in the illustration.
- Of course, use only primitives and wires. See Table 28-1 of IEEE Std 1800-2012 for a list of gates.

Problem 2: Answer the following questions about Verilog primitives as defined in IEEE Std 1800-2012. (See Chapter 28.) Indicate the exact section number where the answer is found.

(*a*) The standard provides a `not` primitive and a `nor` primitive, among others. One can easily argue that a 1-input `nor` gate is the same as a `not` gate. Does the standard actually allow Verilog code to instantiate a 1-input `nor` gate?

(*b*) Based on the standard, is there anything that can be done with a `not` primitive that can't be done with a 1-input `nor` primitive? (Don't try to answer this too deeply, just show an instantiation.)

Problem 3: Output `match` of module `is_1133`, shown below, is `1` iff its input `d` (digits) is 1133 in BCD (which has the same representation as 1133_16). The module instantiates BCD digit detection modules `is_1` and `is_3`.

```
module is_1( output uwire match, input uwire [3:0] d );
   uwire z321;
   nor o0(z321,d[3],d[2],d[1]);
   and a1(match,z321,d[0]);
endmodule

module is_3( output uwire match, input uwire [3:0] d );
   uwire z32;
   nor o0(z32,d[3],d[2]);
   and a1(match,z32,d[1],d[0]);
endmodule

module is_1133( output uwire match, input uwire [15:0] d );
   uwire m1, m2, m3, m4;

   and a1(match, m1, m2, m3, m4);

   is_1 i0(m1, d[15:12]);
   is_1 i1(m2, d[11:8]);
   is_3 i2(m3, d[7:4]);
   is_3 i3(m4, d[3:0]);
endmodule
```

(*a*) Draw a diagram of `is_1133` based on the explicit structural description above. Show the insides of the `is_1` and `is_3` modules. Label the diagram using the same wire and instance names used in the Verilog descriptions.

(*b*) Design a module `is_1133_is` that does the same thing as `is_1133`, but that uses implicit structural code. The correct solution requires adding only one short line to the shell shown below. Don't forget that the value in `d` is in BCD. *Note: The word* short *was added after the original assignment.*

```
module is_1133_is( output uwire match, input uwire [15:0] d );


endmodule
```

Problem 4: When completed the output of module `is_1235` is `1` iff the input is 1235 in BCD.

```
module is_1235( output uwire match, input uwire [15:0] d );

endmodule
```

(*a*) Complete the module. The module must be explicitly structural except for the use of the concatenation operator (see Section 11.4.12). The module **must** use `is_1` and `is_3` to detect the digits. Do not assume or design an `is_2` or `is_5` and don't put in logic to detect those digits.

(*b*) Draw a diagram of the completed module, which should be very similar to the diagram from the previous problem.

LSU EE 4755 Homework 2 Due: 22 September 2016

Problem 0: First, follow the instructions for account setup and homework workflow on the course procedures page, `http://www.ece.lsu.edu/koppel/v/proc.html`.

Look through the code in `hw02.v`. It contains partially completed modules for an ASCII-coded radix-R adder. An overview of ASCII-coded adders and the contents of `hw02.v` is given here in Problem 0, where there is nothing to answer. The problem problems start at Problem 1.

Consider first a hypothetical ASCII-coded radix-10 (decimal) adder with two 5-character ($5 \times 8 = 40$ bit) inputs, and a 5-character output. If the strings `___10` and `__418` appeared at the inputs, the string `__428` should appear at the output (the underscores are supposed to be blanks). The adder could be constructed from 5 *ASCII full adders*, which operate analogously to binary full adders. Each ASCII full adder has two 8-bit inputs, an 8-bit output, and a one-bit carry in and carry out. The value output is the sum of the values at the two inputs plus the carry in. The ASCII full adders should be connected to each other in the same way that binary full adders are connected to make a ripple adder.

BCD and ASCII adders have the following design detail that needs to be decided upon: what to do about input that's not a valid digit. For example, what should the radix-10, ASCII adder do for `__x10` + `__234`? For the adders in this assignment, the decision is to end the number at the first non-digit character starting from the right. So `__x10` would be 10 however `__10_` and `_10y` would both be treated as zero because there first digit character is after non-digit characters (starting from the right).

The modules in this assignment try to use inputs `is_dig_in` and `is_dig_out` to indicate whether there is still a run of digits. (There is a small problem in the implementation, a topic for a future assignment. Anyway, the testbench doesn't test for that.)

Here is a summary of the modules in `hw02.v`:

`aa_decimal_digit_val`: Is complete, don't touch. Determines the binary value and validity of an ASCII decimal digit.

`aa_digit_val`: Incomplete, see Problem 1. Should determine the binary value and validity of a radix-R digit. Tested by the testbench.

`aa_full_adder`: Incomplete, see Problem 2. Should add two radix-R ASCII digits.

`aa_width2`: Is complete, don't touch. A two-digit ASCII-coded, radix-R adder. Instantiates two `aa_full_adder` modules. Tested by the testbench. Will not work correctly when `aa_digit_val` or `aa_full_adder` don't work correctly (which is the initial state of the file).

`reference_adder`: Complete, don't touch. A binary adder with the same range as a 2-digit, radix-R adder. It's purpose is to compare the cost and performance of synthesized hardware.

The modules below are used to implement the testbench. Only modify these to help debug your code.

`radtos`: Convert an integer into a radix-R ASCII string.

`aa_test`: Top-level module for the testbench. It instantiates testbenches for `aa_digit_val` and `aa_full_adder` at each radix from 2 to 16.

`aa_test_digit_val`: Test `aa_digit_val` using every possible input.

`aa_test_width2`: Test `aa_width2` using 100 randomly chosen numbers. These numbers only contain digits.

Run the testbench on the unmodified file. It should report errors for `aa_digit_val` and for `aa_width2`.

Note: There are no points for this problem.

Problem 1: Module `aa_decimal_digit_val`, below, has an 8-bit input `char` and two outputs. Output `is_dig` is 1 iff `char` (an ASCII character) is considered a decimal digit. Output `val` is the value of that digit (in binary), or zero if it's not a digit.

```
module aa_decimal_digit_val
 ( output wire [3:0] val, output wire is_dig, input wire [7:0] char );
   assign       is_dig = char >= "0" && char <= "9";
   assign       val = is_dig ? char - "0" : 0;
endmodule
```

Originally module `aa_digit_val` (see `hw02.v`) is the same as `aa_decimal_digit_val`. Modify `aa_digit_val` so that it honors the value of its `radix` parameter. That is, modify it so that `is_dig` is 1 iff `char` (an ASCII character) is considered a digit in radix `radix` and so that `val` is the value (in binary) of that digit. The module should work correctly for all radices from 2 to 16. For radices ≥ 10 only use lower-case letters for alphabetic digits. Please don't change the width of `val`.

Run the testbench (press F9) to check whether `aa_digit_val` is running correctly and make sure that it is synthesizable.

To check for synthesizability of a module follow the Verilog Synthesis steps given on the procedures page up to and including the elaborate command. There should be no warnings. The synthesis script can be run with the command `rc -files syn.tcl`, it's purpose will be described in the next homework.

Problem 2: When completed module `aa_full_adder` is supposed to add together two digits of a radix-`R` number represented in ASCII plus a carry in. Output `sum` of the module is the ASCII digit of the sum, and output `carry_out` is `1` iff there is a carry.

Complete module `aa_full_adder` so that it operates as described. The module should instantiate two `aa_digit_val` modules and use them to generate the sum digit. The module must be synthesizable, it can be written using implicit structural or behavioral code.

Run the testbench to verify correct functioning.

To check for synthesizability of a module follow the Verilog Synthesis steps given on the procedures page up to and including the elaborate command. There should be no warnings. The synthesis script can be run with the command `rc -files syn.tcl`, its purpose will be described in the next homework.

LSU EE 4755 Homework 3 Due: 28 September 2016

Problem 1: Module `aa_digit_val`, below, is the solution to Homework 2 Problem 1. It has an 8-bit input `char` and two outputs. Output `is_dig` is 1 iff `char` (an ASCII character) is considered a radix-R digit, where $2 \leq R \leq 16$, is the value of parameter `radix`. Output `val` is the value of that digit (in binary), or zero if it's not a digit.

```
module aa_digit_val
  #( int radix = 10 )
   ( output uwire [3:0] val,   output uwire is_dig,   input uwire [7:0] char );

   uwire  is_dig_09 = char >= "0" && char <= "9";
   uwire  is_dig_af = char >= "a" && char <= "f";
   uwire [3:0] val_raw = is_dig_09 ? char - "0" : char - "a" + 10;
   assign     is_dig = ( is_dig_09 || is_dig_af ) && val_raw < radix;
   assign     val = is_dig ? val_raw : 0;

endmodule
```

Provide sketches of what you expect the inferred hardware to look like for `aa_digit_val` as described below. *Hint: Some problems in the EE 4755 2014 Final Exam dealt with numbers in ASCII representation. The optimizations requested below must go beyond those found in the exam solution.*

(*a*) Show a sketch of the inferred hardware before any optimization is done.

(*b*) Show a sketch of the inferred hardware after some optimization has been performed.

- The sketches must show the product of human thought (in particular, the human who's name is on the submission), not a synthesis program.
- When considering the optimizations for the logic generating `is_dig` (including the logic for `is_dig_09` and `is_dig_af`) recall that in general the cost of logic computing `a==b` is less than the cost of logic computing `a>b`.
- When considering the optimizations for the logic generating `val` think about the subtraction operations and what they actually do when `is_dig` is true. If necessary, work out examples of the subtraction by hand in hexadecimal.

There is another problem on the next page!

Problem 2: Module `aa_full_adder` from Homework 2, Problem 2 adds together two digits of a radix-`R` number represented in ASCII plus a carry in. The module description from the solution appears below.

```
module aa_full_adder
  #( int radix = 10 )
   ( output uwire [7:0] sum, output uwire carry_out, output uwire is_dig_out,
     input uwire [7:0] a, b, input uwire carry_in, input uwire is_dig_in);

   uwire [3:0] val_a, val_b;
   uwire       is_dig_a, is_dig_b;

   aa_digit_val #(radix) dva(val_a, is_dig_a, a);
   aa_digit_val #(radix) dvb(val_b, is_dig_b, b);

   assign is_dig_out = is_dig_in && ( carry_in || is_dig_a || is_dig_b );
   uwire [4:0] sum_val = carry_in + val_a + val_b;
   assign     carry_out = sum_val >= radix;
   uwire [3:0] sum_dig_val = carry_out ? sum_val - radix : sum_val;
   assign sum = !is_dig_out ? " " :
          sum_dig_val < 10  ? "0" + sum_dig_val : "a" + sum_dig_val - 10;

endmodule
```

An obvious objection to an ASCII-coded radix-R adder is that it uses 8 bits to represent a digit that can be represented using only $\lceil \lg R \rceil$ bits.

(a) Show the hardware that might be synthesized for the module `aa_full_adder` based on the description above. This should be the inferred hardware with some optimizations applied. Take care to show the number of bits at the inputs and output of units like adders and comparison logic.

(b) Compare the cost of a d-digit ASCII-coded radix-16 adder to a $4d$-bit ripple adder. (Note that both adders can add numbers in the range of 0 to $2^{4d} - 1$.) Do so by estimating the cost in terms of the number of gates, and state any assumptions, such as the number of gates needed for an x-bit comparison unit.

LSU EE 4755 Homework 4 Due: 12 October 2016

Problem 0: First, follow the instructions for account setup and homework workflow on the course procedures page, `http://www.ece.lsu.edu/koppel/v/proc.html`.

Look through the code in `hw04.v`. Module `lookup_behav` in file `hw04.v` has a w-bit input `char` and an n-element array of w-bit quantities named `chars`. (Parameter `nelts` is n and parameter `charsz` is w.) The module also has a 1-bit output `found` which is logic `1` iff any element of `chars` is equal to `char`. Finally, the module has a $\lceil \lg n \rceil$-bit output `index` which is set to the element number of `chars` that matches `char`, or `0` if `found` is `0`. Assume that no two elements of `chars` are identical.

For example, suppose input `char` is set to `102` and that `chars` is `{63,124,102,92}`. Then output `found` will be `1` and `index` will be `2`. If `char` were `7` `index` would be `0` and `found` would be `0`, if `char` were `63` `index` would be `0` and `found` would be `1`, etc. The alert student will have recognized that $n = 4$ and that $w \geq 7$ in these examples.

Module lookup is coded in synthesizable behavioral form that describes combinational logic. The `hw04.v` file contains two other modules which are to do the same thing, `lookup_linear` and `lookup_tree`, but those modules are not yet finished.

The testbench tests all of these modules. It tests them for sizes (n) of 4, 5, 10, 15, 16, 30, 40, and 64. To change which sizes are tested (or the order in which they are tested) edit the `testbench` module.

To have the testbench test only some of these modules (say, skip the `lookup_tree` tests until after `lookup_linear` is working) look for the `for` loop with `mut=0` and modify it appropriately. (It should be easy to figure out the numbers.)

A synthesis script is provided that will synthesize all three modules at different sizes and both with and very lax timing constraint and a very strict timing constraint. The script can be run using the command `rc -files syn.tcl`. Initially it will stop with an error. To see it run to completion before starting the assignment have it only synthesize `lookup_behav` (see below). Pre-set synthesis options (in file `.synth_init`) were chosen to reject any design that is not combinational.

If there is an error when using the synthesis script then follow the manual synthesis steps on the procedures page and look for error messages.

To change which modules are synthesized edit the `set modules` line (near the bottom) in file `syn.tcl`. The values for `nelts` and other items can also be changed by editing the file.

Note: There are no points for this problem.

Problem 1: Complete `lookup_linear` so that it does the same thing as `lookup_behavioral` but by using as many copies of `lookup_elt` as it needs. That is, `lookup_linear` should use generate statements to instantiate `lookup_elt` and it should include whatever other code is needed to use these instances to compute the correct outputs.

- Behavioral or structural code can be used.
- The module must be synthesizable.
- Assume that all elements of `chars` are different.

Problem 2: Complete module `lookup_tree` so that it performs the lookup using recursive instantiations of itself. Take care so that `index` is computed efficiently. *Hint: think about how to compute index efficiently when n (*`nelts`*) is a power of 2, then get the same efficiency for any n.*

If completed correctly, the cost and especially the performance at larger sizes should be better than `lookup_behavioral` and (unless you did an unexpectedly good job) better than `lookup_linear`.

- Behavioral or structural code can be used.
- The module must be synthesizable.
- Assume that all elements of `chars` are different.

Problem 3: Run the synthesis script and characterize the strengths and weaknesses of each module. (For example, module X has lowest cost for low-speed designs.)

In a follow-on homework assignment additional questions will be asked about these modules.

LSU EE 4755 Homework 5 Due: 7 November 2016

Problem 0: This first problem provides background on the module used in this assignment. Please read the background and then solve the problems further below. The Verilog source can be found in directory `hw05`, however for this assignment there is no need to do anything with it.

Module `ortho` has one input, `v`, a three-element vector of signed integers, and one output, `u`, also a three-element vector of signed integers. The output is computed so that `u` is orthogonal to `v` in the geometric sense. For those who are rusty on linear algebra, non-zero vectors u and v are orthogonal if $u \cdot v = 0$ or $u_x v_x + u_y v_y + u_z v_z = 0$. Using Verilog notation, `u` is computed so that `u[0]*v[0]+u[1]*v[1]+u[2]*v[2]=0` and at least one element of `u` is not zero. It does so by finding the smallest element of `v`, setting the corresponding element in `u` to zero, swapping the to remaining two elements, and negating one of the two. For example, if $v = (4, 7, 55)$ then the module would set $u = (0, 55, -7)$.

```
module ortho #( int alternative = 1, int w = 32 )
   ( output logic signed [w-1:0] u [3],   input wire signed [w-1:0] v [3] );

   logic [1:0] idx_min, idx_a, idx_b;

   always_comb begin

      idx_min = 0;
      for ( int i=1; i<3; i++ )
        if ( $abs(v[i]) < $abs(v[idx_min]) ) idx_min = i;

      idx_a = ( idx_min + 1 ) % 3;
      idx_b = ( idx_min + 2 ) % 3;

      if ( alternative == 1 ) begin

         // The loop below is needed as a hint to the synthesis program
         // Cadence Encounter 14.28.
         for ( int i=0; i<3; i++ ) u[i] = 0;

         u[idx_min] = 0;
         u[idx_a] = v[idx_b];
         u[idx_b] = -v[idx_a];

      end else if ( alternative == 2 ) begin

         for ( int i=0; i<3; i++ )
           u[i] = idx_min == i ? 0 : idx_a == i ? v[idx_b] : -v[idx_a];

      end else $fatal(1);

   end

endmodule
```

Important: For all problems below in which hardware is shown:

- Clearly show inputs and outputs of `ortho`.
- Try to draw diagrams showing all hardware for `ortho` and refer to parts of the diagram in your answers below.

Problem 1: Consider the following part of the module:

```
idx_min = 0;
for ( int i=1; i<3; i++ )
  if ( $abs(v[i]) < $abs(v[idx_min]) ) idx_min = i;
```

(*a*) Show the hardware that will be synthesized for this fragment. (Please refer to the entire module when determining what will be synthesized.) Make reasonable optimizations. (See the next subproblem.) In this subpart show `abs` as a box.

(*b*) The synthesis program synthesizes hardware that contains four absolute value units for this code, even with effort set to high. Explain why four is too many, perhaps by referring your own version that uses fewer absolute value units.

Problem 2: Consider the part of the module below: Show the hardware that will be synthesized for this code, taking into consideration that `idx_min` is two bits. *Hint: This is easy. Just consider all possible values of* `idx_min`.

```
idx_a = ( idx_min + 1 ) % 3;
idx_b = ( idx_min + 2 ) % 3;
```

Problem 3: Show the hardware that will be synthesized for the alternative 2 code, below, after optimization. As with the other problems, take into account the rest of the module. Look for opportunities to optimize `-v[idx_a]` taking advantage of hardware for `abs`.

```
for ( int i=0; i<3; i++ )
  u[i] = idx_min == i ? 0 : idx_a == i ? v[idx_b] : -v[idx_a];
```

Problem 4: As directed below, estimate the critical path in `ortho` for a w-bit instantiation. Do so using ripple-adder like implementations for absolute value, comparison, and negation. Use the performance model in which n-input AND and OR gates have delay $\lceil \lg n \rceil$ units.

(*a*) Find the critical path using the assumption that in hardware for an expression like $a + b < c$ the delay through the adder must be added to the delay through the comparison unit. The answer should be a function of w.

(*b*) Find the critical path accounting for the fact that in ripple-like hardware for an expression like $a + b < c$ the low bits of the comparison can start as soon as the low bits of the sum are available. The answer should be a function of w.

(*c*) Show a sketch of the hardware with an arrow tracing the critical path through the hardware, from input to output. Annotating that arrow with intermediate delays will help in assigning partial credit.

LSU EE 4755 Homework 6 Due: 29 November 2016

Problem 0: Review the instructions for account setup and homework workflow on the course procedures page, `http://www.ece.lsu.edu/koppel/v/proc.html`.

Look through the code in `hw06.v`. These modules compute the floating-point sum of squares of their input, similar to the midterm exam problem but without the square root.

Module `mag_functional` is a non-synthesizable version of the module. It is not synthesizable by Cadence Encounter because it operates on floating-point values. The module is included to help in understanding the other modules.

Module `mag_comb` is a synthesizable combinational version of the module. The floating-point operations are implemented using modules from the ChipWare library. See the ChipWare documentation, linked to the course references page, for details.

Module `mag_seq`, when finished, computes mag sequentially. It contains some code, including floating-point module instantiations, but is not complete. It has an input `start` to initiate the computation and an output `ready` to signal that the computation is complete.

Module `mag_pipe`, when finished, computes mag in pipelined fashion. At each positive edge it reads a vector from its input and provides the mag of a prior vector at its output.

Module `mag_comb` should be fastest, but of high cost. Module `mag_seq` should be the lowest cost module and `mag_pipe` should be the highest cost but also the highest throughput.

The testbench provides test inputs to the three synthesizable modules. Initially, `mag_comb` should pass all tests and the others should fail all tests. To facilitate debugging the first eight tests are the vectors $[0,0,0], [0,0,1], [0,1,0], [0,1,1], \ldots$. After that the vector components are randomly chosen over the range $[-10, 10]$.

Remember that the values are IEEE 754 single-precision floating point. A 0 in this FP representation is `32'h0` and a 1.0 is `32'h3f800000`, a 2.0 is `32'h40000000`, and a 3.0 is `32'h40400000`.

To solve this assignment it is very important to use the waveform viewer for debugging. To do so start the simulator graphically using the command `irun -gui hw06.v`. From the Design Browser pane locate `testbench` and under it look for `m2` (for `mag_seq`) or `m3` (for `mag_pipe`). Select objects in the Objects pane and send them to the waveform window by pressing the waveform toolbar button (it looks like a logic analyzer display) or by selecting the sequence Windows → Send To → Waveform. Run the simulator by pressing the play toolbar icon. If you've made changes to the Verilog or otherwise want to re-run the simulation without exiting select Simulation → Reinvoke Simulator.

For further documentation see the SimVision documentation on the course references page, `http://www.ece.lsu.edu/koppel/v/ref.html`.

There is a synthesis script that will synthesize each module at high (slow) and low (fast) clock period targets. To run it use the command `rc -files syn.tcl`. This will take a long time to run, so only run it to satisfy your curiosity. Check for synthesizability by manually running the synthesis using the instructions on the course procedures page, `http://www.ece.lsu.edu/koppel/v/proc.html`.

Note: There are no points for this problem.

Problem 1: Module `mag_seq`, when completed, will compute the magnitude sequentially. It should start when input `start` is logic 1 on a positive clock edge and it should signal completion by setting output `ready` to one several clock cycles later. The module should use one floating-point multiply and one FP add unit. The module already instantiates these, and contains some logic for performing the different steps, including setting `ready` (though not at the right time). Complete the module so that it works correctly. See the checklist in the Verilog source for hints and reminders.

Problem 2: Module `mag_pipe`, when completed, will compute the magnitude in pipelined fashion. That is, it will read a vector from its inputs at every clock cycle, and will present a magnitude at its output every cycle. The magnitude should be for the vector that was at input `v` `nstages` cycles in the past, where `nstages` is a constant in the module indicating the number of stages. The inputs to the module are available near the end of the clock cycle and the outputs are expected at the beginning of the clock cycle.

Choose the number of stages needed to maximize throughput. That is, minimize the delay in each stage. Of course, within that constraint minimize cost.

Pay close attention to where data is. Remember that at any one time the module will hold data for `nstages` different vectors. Use a pipeline diagram to make sure that data from the different vectors don't get mixed up. A common problem is a newly arriving vector overwriting data for an earlier vector. That's avoided by moving data long from stage to stage.

Be sure to use the waveform viewer for debugging. Remember that the first eight test vectors consists of 0 and 1 components, making debugging easy.

10 Fall 2015

LSU EE 4755 Homework 1 Due: 9 September 2015

The questions below can be answered without using EDA software, paper and pencil will suffice. Please turn in the solution on paper. Homework 2 will require the use of Verilog implementations.

Those who are rusty about the correspondence between Verilog code and hardware might want to look at the solution to EE 3755 Fall 2013 Homework 1, at `http://www.ece.lsu.edu/ee3755/2013f/hw01_sol.pdf`.

Problem 1: The routine `shift_right_fixed_amt` uses the `>>` operator to perform the right shift. Perhaps you are wondering if the operation is an arithmetic right shift or a logical right shift. (In a logical right shift the vacated bit positions are always set to zero, in an arithmetic shift they are set to the MSB of the input.) Look up the operation performed by this operator in the SystemVerilog 2012 documentation.

```
module shift_right_fixed_amt
  #( int fsamt = 4 )   // Fixed shift amount.
   ( output wire [15:0] shifted,
     input wire [15:0] unshifted,
     input wire shift );

   // If shift is true shift by fsamt, otherwise don't shift.
   //
   assign    shifted =  shift  ?  unshifted >> fsamt  :  unshifted;

endmodule
```

(*a*) Indicate the section and page in which this information can be found.

(*b*) Show how the module can be modified to perform the other kind of shift (if it's currently arithmetic, make it logical, if it's currently logical make it arithmetic).

Problem 2: Appearing below are two variations on a `min_4` module that finds the minimum of four unsigned integers. Both of these modules instantiate the following `min_2` module.

```
module min_2
  #( int elt_bits = 4 )
   ( output [elt_bits-1:0] elt_min,
     input [elt_bits-1:0] elt_0,
     input [elt_bits-1:0] elt_1 );
   assign                 elt_min = elt_0 < elt_1 ? elt_0 : elt_1;
endmodule
```

(*a*) Draw a diagram of the hardware that will be synthesized for the `min_4_t` module below. Your diagram should include two-input multiplexors and a comparison module. To get an idea of what to draw, see the EE 3755 Homework solution mentioned at the top of this assignment.

```
module min_4_t
  #( int elt_bits = 4 )
   ( output [elt_bits-1:0] elt_min,
     input [elt_bits-1:0] elts [4] );

   wire [elt_bits-1:0]    im1, im2;
   min_2 #(elt_bits) m1( im1, elts[0], elts[1] );
   min_2 #(elt_bits) m2( im2, elts[2], elts[3] );
   min_2 #(elt_bits) m3( elt_min, im1, im2 );
endmodule
```

(*b*) Draw a diagram of the hardware that will be synthesized for the `min_4_l` module below. Your diagram should include two-input multiplexors and a comparison module.

```
module min_4_l
  #( int elt_bits = 4 )
   ( output [elt_bits-1:0] elt_min,
     input [elt_bits-1:0] elts [4] );

   wire [elt_bits-1:0]    im1, im2;
   min_2 #(elt_bits) m1( im1, elts[0], elts[1] );
   min_2 #(elt_bits) m2( im2, im1, elts[2] );
   min_2 #(elt_bits) m3( elt_min, im2, elts[3] );
endmodule
```

(*c*) Which of the two modules above would you expect to have lower cost? Which would you expect to be faster? Briefly explain.

Problem 3: The module `min_4_err` below is correct Verilog, but it won't do what we want.

```
module min_4_err
  #( int elt_bits = 4 )
   ( output [elt_bits-1:0] elt_min,
     input [elt_bits-1:0] elts [4] );

   wire [elt_bits-1:0]    im;
   min_2 #(elt_bits) m1( im, elts[0], elts[1] );
   min_2 #(elt_bits) m2( im, im, elts[2] );
   min_2 #(elt_bits) m3( elt_min, im, elts[3] );

endmodule
```

(*a*) Explain why it's correct Verilog yet provides the incorrect result.

(*b*) Look up `uwire` in the SystemVerilog standard and explain how that might help catching such errors.

Problem 4: Appearing below is yet another variation on `min_4`, this one attempting to take advantage of a special case by using generate statements. The module is correctly using generate statements to handle a special case. Do you think the synthesized hardware will be less expensive for the special case *beyond the reduction in cost for using fewer bits*. Hint: Think about what the comparison unit and mux would look like with 1-bit inputs and how such logic can be optimized.

Note: In the original assignment this problem had a typo, which made the Verilog illegal. Further, the phrase above starting "beyond the reduction" was not in the original question, making it difficult to see what was really being asked. The answer below is for the corrected question.

```
module min_4_special1
  #( int elt_bits = 4 )
   ( output [elt_bits-1:0] elt_min,
     input [elt_bits-1:0] elts [4] );

   if ( elt_bits == 1 ) begin

      assign elt_min = elts[0] && elts[1] && elts[2] && elts[3];

   end else begin

      wire [elt_bits-1:0]    im1, im2;

      min_2 #(elt_bits) m1( im1, elts[0], elts[1] );
      min_2 #(elt_bits) m2( im2, im1, elts[2] );
      min_2 #(elt_bits) m3( elt_min, im2, elts[3] );

   end

endmodule
```

Problem 5: The module below handles another special case, in this case the case where the first element is zero.

```
module min_4_special2
  #( int elt_bits = 4 )
   ( output [elt_bits-1:0] elt_min,
     input [elt_bits-1:0] elts [4] );

   wire [elt_bits-1:0]    im1, im2;

   if ( elts[0] == 0 )
     assign elt_min = 0;
   else begin
      min_2 #(elt_bits) m1( im1, elts[0], elts[1] );
      min_2 #(elt_bits) m2( im2, im1, elts[2] );
      min_2 #(elt_bits) m3( elt_min, im2, elts[3] );
   end
endmodule
```

(*a*) Explain why the module is illegal Verilog.

(*b*) Explain why what it's trying to do would be unlikely to help within a larger design. *Hint: Think about critical path.*

LSU EE 4755 Homework 2 Due: 16 September 2015

Problem 0: Follow the instructions for account setup and homework workflow on the course procedures page, `http://www.ece.lsu.edu/koppel/v/proc.html`. Run the testbench on the unmodified file. There should be errors on all but the `min_4` (Four-element) module. Try modifying `min_4` so that it simulates but produces the wrong answer. Re-run the simulator and verify that it's broken. Then fix it.

Note: There are no points for this problem.

Problem 1: Module `min_n` has an `elt_bits`-bit output `elt_min` and an `elt_count`-element array of `elt_bits`-bit elements, `elts`. Complete `min_n` so that `elt_min` is set to the minimum of the elements in `elts`, interpreting the elements as unsigned integers. Do so using a linear connection of `min_2` modules instantiated with a `genvar` loop. (A linear connection means that the output of instance i is connected to the input of instance $i + 1$.)

Verify correct functioning using the testbench.

Problem 2: Module `min_t` is to have the same functionality as `min_n`. Complete `min_t` so that it recursively instantiates itself down to some minimum size. The actual comparison should be done by a `min_2` module.

Verify correct functioning using the testbench.

Problem 3: By default the synthesis script will synthesize each module for two array sizes, four elements and eight elements.

(a) Run the synthesis script unmodified. Use the command `rc -files syn.tcl`. Explain the differences in performance between the different modules.

(b) Modify and re-run the synthesis script so that it synthesizes the modules with `elt_bits` set to 1.

The synthesis program should do a better job on the behavioral and linear models. Why do you think that is? *Hint: The 1-bit minimum module is equivalent to another common logic component that the synthesis program can handle well.*

LSU EE 4755 Homework 3 Due: 7 October 2015

Problem 1: Solve EE 4755 Fall 2014 Midterm Exam Problem 4 and Problem 5. The solutions are available, but please make an honest effort to solve them on your own.

Problem 2: The homework Verilog file, `hw04.v` contains two versions of the sequential shifter used in class, those modules are also reproduced below. Module `shift_lt_seq_d_live`, is based on the version written during class and module `shift_lt_seq_d` is the one prepared in advance. Though both work correctly their timing is not identical.

(*a*) Show the hardware that might be synthesized for each module using the default parameters. Include reasonable optimizations, the initially inferred hardware can be omitted. This should be a human-to-human diagram, don't show output of a synthesis program.

(*b*) The two modules differ in their timing. Using your hardware diagrams explain any differences in:

- The register-to-register delay within the module.
- How far in advance of the positive edge module inputs must become stable.
- How long after the positive edge module outputs will be available.

As with the previous part, this should be done by hand though synthesis tools can be used to help solve the problem.

An answer might look like this: *"For register-to-register delay Module A is slower because its critical path has two multipliers, whereas in module B the two multiplications are split between cycles and so at most one multiplier is on the critical path. In module A inputs connect directly to a divider, and so they must arrive long before the positive edge, whereas in module B inputs can arrive just before the positive edge because ...."* Of course, this question does not have a module A or B, nor does it really have multipliers and dividers.

Modules on next page.

```
module shift_lt_seq_d_live
  #( int wid_lg = 6,
     int num_shifters = 1,
     int wid = 1 << wid_lg )
   ( output logic [wid-1:0] shifted,
     output logic ready,
     input [wid-1:0] unshifted,
     input [wid_lg-1:0] amt,
     input start,
     input clk );

   localparam int bits_per_seg = wid_lg / num_shifters;

   logic [num_shifters-1:0] shift;
   wire [wid-1:0] shin[num_shifters-1:-1];
   assign shin[-1] = shifted;

   for ( genvar i=0; i<num_shifters; i++ ) begin
      localparam int fs_amt = 2 ** ( i * bits_per_seg );
      shift_fixed #( wid_lg, fs_amt ) sf( shin[i], shin[i-1], shift[i] );
   end

   logic [num_shifters-1:0][bits_per_seg-1:0] cnt;

   always_ff @( posedge clk ) begin

      if ( start == 1 ) begin
         ready = 0;
         cnt = amt;
         shift = 0;
         shifted = unshifted;
      end else begin
         if ( cnt == 0 ) ready = 1;
         for ( int i=0; i<num_shifters; i++ ) begin
            shift[i] = cnt[i] > 0;
            if ( cnt[i] != 0 ) cnt[i]--;
         end
         shifted = shin[num_shifters-1];
      end

   end

endmodule
```

Another module on next page.

```
module shift_lt_seq_d
  #( int wid_lg = 4,
     int num_shifters = 2,
     int wid = 1 << wid_lg )
   ( output logic [wid-1:0] shifted,
     output wire ready,
     input [wid-1:0] unshifted,
     input [wid_lg-1:0] amt,
     input start,
     input clk );

   localparam int cnt_bits = ( wid_lg + num_shifters - 1 ) / num_shifters;
   logic [num_shifters-1:0][cnt_bits-1:0] cnt;
   wire [wid-1:0] inter_sh[num_shifters-1:-1];
   assign inter_sh[-1] = shifted;

   for ( genvar i = 0; i < num_shifters; i++ ) begin
      localparam int shift_amt = 1 << i * cnt_bits;
      wire       shift = cnt[i] != 0;
      shift_fixed #(wid_lg,shift_amt) sf( inter_sh[i], inter_sh[i-1], shift );
   end

   always_ff @( posedge clk )

      if ( start == 1 ) begin
         shifted = unshifted;
         cnt = amt;
      end else if ( cnt > 0 ) begin
         shifted = inter_sh[num_shifters-1];
         for ( int i=0; i<num_shifters; i++ ) if ( cnt[i] ) cnt[i]--;
      end

   assign ready = cnt == 0;

endmodule
```

LSU EE 4755 Homework 4 Due: 12 October 2015

Problem 0: Follow the instructions for account setup and homework workflow on the course procedures page, `http://www.ece.lsu.edu/koppel/v/proc.html`. Run the testbench on the unmodified file. There should be errors on the `shift_lt_seq_d_sol` module, but the others should run correctly. Run the Note: There are no points for this problem.

Problem 1: The homework Verilog file, `hw04.v`, contains a module `shift_lt_seq_d_sol` which is based on `shift_lt_seq_d`. It contains an `always_ff` block that assigns the same variables that are assigned in `shift_lt_seq_d`, however it assigns them from variables of the same name with `next_` prefixed:

```
always_ff @( posedge clk ) begin
   ready = next_ready;
   shifted = next_shifted;
   shift = next_shift;
   cnt = next_cnt;
end
```

Add code so that these `next_` objects will be assigned values from combinational logic, and so that the resulting module describes the same hardware as `shift_lt_seq_d`. A hand-drawn diagram of synthesized hardware should be identical, though it's possible that there will be small differences in the actual output of a synthesis program.

The added code can be implicit structural or behavioral, but it must synthesize to combinational logic.

Problem 2: Module `shift_lt_seq_d_live` takes one more cycle to produce a result than module `shift_lt_seq_d`. Module `shift_lt_seq_d_p2` initially is identical to `shift_lt_seq_d_live`.

(*a*) Modify `shift_lt_seq_d_p2` so that it uses one less cycle to produce a result without changing the number of shifters per stage. There are two possible ways of doing this, performing some work in the same cycle that the `start` signal arrives, or doing work in the cycle when `ready` is set to 1. Either method is fine.

(*b*) Run `syn.tcl` and compare the cost and performance of your design and `shift_lt_seq_d_live`. Comment on the differences. An answer might start *"The cost was about the same because the same hardware was used..."*.

LSU EE 4755 Homework 5 Due: 23 October 2015 17:00

Problem 1: The homework Verilog file, `hw05.v`, contains something similar to the streamlined multiplier presented in class, `mult_seq_stream`, and even more streamlined versions of the multiplier, `mult_seq_stream_2`, and `mult_seq_stream_3`. These modules are reproduced at the end of this assignment. For an HTML version visit
`http://www.ece.lsu.edu/koppel/v/2015/hw05.v.html`. *See the 2014 midterm exam for similar problems.*

(*a*) Show the hardware that will be synthesized for each module for the default parameters. Show the module after optimization.

(*b*) Estimate the clock frequency of each module based on the following assumptions:

Latch delay: 10 units. Multiplexor latency: 2 units. Latency of a n-bit adder: $5\lceil \lg n \rceil$ units. Latency of an n-input gate: $\lceil \lg n \rceil$ units.

(*c*) Why would module `mult_seq_stream_3` provide a result in less time than the other two, even assuming that the clock frequency for all the modules was the same?

```
module mult_seq_stream #( int wid = 16 )
   ( output logic [2*wid-1:0] prod,
     input logic [wid-1:0] plier,
     input logic [wid-1:0] cand,
     input clk);

   localparam int wlog = $clog2(wid);

   logic [wlog-1:0] pos;
   logic [2*wid-1:0] accum;

   always @( posedge clk ) begin

      logic [wid:0] pp;

      if ( pos == 0 ) begin

         prod = accum;
         accum = cand;
         pos = wid - 1;

      end else begin

         pos--;

      end

      // Note: the multiplicand is in the lower bits of the accumulator.
      //
      pp = accum[0] ? { 1'b0, plier } : 0;

      // Add on the partial product and shift the accumulator.
      //
      accum = { { 1'b0, accum[2*wid-1:wid] } + pp, accum[wid-1:1] };

   end

endmodule
```

```
module mult_seq_stream_2 #( int wid = 16 )
   ( output logic [2*wid-1:0] prod,
     input logic [wid-1:0] plier,
     input logic [wid-1:0] cand,
     input clk);

   localparam int wlog = $clog2(wid);

   logic [wlog-1:0] pos;
   logic [2*wid-1:0] accum;

   always @( posedge clk ) begin

      if ( pos == 0 ) begin

         prod = accum;
         accum = { 1'b0, cand[0] ? plier : wid'(0), cand[wid-1:1] };
         pos = wid - 1;

      end else begin

         logic [wid:0] pp;

         // Note: the multiplicand is in the lower bits of the accumulator.
         //
         pp = accum[0] ? plier : 0;

         // Add on the partial product and shift the accumulator.
         //
         accum = { { 1'b0, accum[2*wid-1:wid] } + pp, accum[wid-1:1] };

         pos--;

      end

   end

endmodule
```

```
module mult_seq_stream_3 #( int wid = 16 )
   ( output logic [2*wid-1:0] prod,
     input logic [wid-1:0] plier,
     input logic [wid-1:0] cand,
     input clk);

   localparam int wlog = $clog2(wid);

   logic [wlog-1:0] pos;
   logic [2*wid-1:0] accum;

   always @( posedge clk ) begin

      if ( pos == 0 ) begin

         accum = { 1'b0, cand[0] ? plier : wid'(0), cand[wid-1:1] };
         pos = wid - 1;

      end else begin

         logic [wid:0] pp;

         // Note: the multiplicand is in the lower bits of the accumulator.
         //
         pp = accum[0] ? plier : 0;

         // Add on the partial product and shift the accumulator.
         //
         accum = { { 1'b0, accum[2*wid-1:wid] } + pp, accum[wid-1:1] };

         if ( pos == 1 ) prod = accum;

         pos--;

      end

   end

endmodule
```

LSU EE 4755 Homework 6 Due: 2 December 2015

Problem 0: The homework Verilog file, `hw06.v`, contains something similar to the integer compression modules presented in class. (Follow the homework workflow instructions on the course procedures page to get a copy of the assignment package.) These modules compress an ASCII character stream by substituting a binary-encoded integer for a string of ASCII digits. These modules were based on 2014 Homework 4. Feel free to look at that assignment an solution for help.

Module `icomp_none` is a version of the module that does no compression at all. It does though implement the handshaking protocol so that characters can be passed from input to output. This module can be studied to help understand how the others work.

Module `icomp_2cyc` is one of the compression modules covered in class. It computes the encoded value in stage 0, and checks for overflow in stage 1. Don't modify this module, save if for reference. Module `icomp_sol` is initially identical to `icomp_2cyc`, but it should be modified as part of this assignment.

The testbench is set to simulate `icomp_sol` on a sample test string. At the end it will report the amount of compression and whether there was any errors. The testbench also prints out a trace showing some module inputs and outputs and the status of internal signals. Examine the testbench code to see how this is done and feel free to modify it to add signals of your own. A more detailed trace of execution can be obtained using the SimVision gui. To start that use the command `irun hw06.v -gui`. See `http://www.ece.lsu.edu/koppel/v/v/s/SimVisionIntro.pdf` for documentation. (On campus access only without password.)

The synthesis script will synthesize the modules `icomp_2cyc` and `icomp_sol`. Use the synthesis script to make sure that your designs are synthesizable and to determine their cost and performance.

(There is nothing to turn in for this assignment.)

Problem 1: In module `icomp_sol` there is a declaration of a variable named `val_encode_size_1`, but no uses of that variable. Add code to that module so that `val_encode_size_1` is set to the number of bytes that are needed for the number currently in the register `val_encode_1`. For example, if `val_encode_1` has a 0, then `val_encode_size_1` should be 0. If `val_encode_1` has a 123 then `val_encode_size_1` should be 1 (one byte), if `val_encode_1` has a 300 then `val_encode_size_1` should be 2 (for 2 bytes), etc.

To help with your solution add code to the testbench to show the value of this variable.

Problem 2: Modify module `icomp_sol` so that a group of ASCII digits is compressed into the smallest number of bytes needed, up to `max_chars`. For example, if `max_chars` is 4 then just use one byte to compress 200, two bytes for 4000, and for 1234567890123 use a four-byte integer (for 1234567890) followed by a one byte integer (for 123).

Precede the compressed integer by the character 128 plus the number of bytes in the compressed number. For example, if the compressed value takes two bytes then where the first character of the uncompressed value would go emit a 130, then the next two characters should be the compressed number. (See how `char_out` is assigned in the unmodified code.)

To solve this problem you'll need to understand how the existing code works, how to interpret the trace output provided by the simulator, and how to use the SimVision waveform viewer. Random guesses based on a vague understanding will get you nowhere.

- The module should be written for arbitrary values of `max_chars`.
- Make sure that the testbench is not reporting errors.

- Make sure that your module is compressing the string.

11 Fall 2014

LSU LSU EE 4755 Homework 1 Due: 15 September 2014

Follow the instructions for class account setup and Verilog Homework Workflow, which can be found on `http://www.ece.lsu.edu/koppel/v/proc.html`. *Run the simulator code on the unmodified assignment. The output should show errors for two modules.*

Problem 1: Module `shift_right1` is supposed to perform a logical right shift on a 16-bit quantity, but it is not working properly, perhaps because the designer left for a vacation before finishing it and returned thinking that he or she had already finished it. Fix the problem.

Module `shift_right1` is written in a behavioral style, and in a way which is not synthesizable. For this problem, **do not** try to make the code synthesiable, just get the module to perform the shift properly so that the testbench does not report an error. (The module for the next problem is synthesizable.)

Your solution should assign `shifted` one bit at a time, as does the existing code. (In other words, don't just use the right shift operator.) The testbench output might provide clues to what the problem is. *Hint: The problem can be fixed with one or two lines of code.*

Problem 2: Module `shift_right2` is also supposed to perform a logical right shift. It's not working either, because it hasn't been finished. When finished `shift_right2` will make use of four `shift_right_fixed` modules. A `shift_right_fixed` module can shift by two possible amounts, zero bits (which of course is no shift at all) or `fsamt` bits, where `fsamt` is the value of a parameter.

The `shift_right2` module so far has instantiated one `shift_right_fixed` module and set the parameter to 8 (the `#(8)` indicates that the parameter is set to 8). The `shift_right2` module should instantiate three more `shift_right_fixed` modules, one each for shifts of 4, 2, and 1 bit. Instantiate the modules and connect them together so that `shift_right2` works correctly.

Hint: A correct answer will require no additional logic (beyond the three additional shifters) only declarations.

LSU EE 4755 Homework 2 Due: 26 September 2014

The Homework 2 code package contains four unsigned integer floating point modules and a testbench. The first two modules, `mult_behav_1` *and* `mult_behav_2` *already work, the other two,* `mult_linear` *and* `mult_tree`*, are mostly empty and are to be completed as part of this assignment. The first two multipliers are synthesizable, though they were not written to be synthesized. If this assignment is completed correctly the other two multipliers will be synthesizable too.*

Multiplier `mult_behav_1` *is a simple-as-possible implementation, the intent is to provide a correct result to use to check the other modules. Nevertheless it is synthesizable with Cadence RC, which will substitute an integer multiply library function from the ChipWare library.*

Multiplier `mult_behav_2` *computes the multiplication itself by adding partial sums. (See* `http://www.ece.lsu.edu/ee3755/2013f/l07.v.html` *for a quick review of integer multiplication. Don't go beyond the long-hand procedure for this assignment.)*

Warning: DO NOT attempt to find Verilog code for multipliers and use them for the solution. You will learn nothing by doing so and will be unprepared for the midterm exam.

Problem 0: Copy the code package from `/home/faculty/koppel/pub/ee4755/hw/2014f/hw02`. Verify that everything is working by running the simulation on the unmodified file. It should report a 0% error rate for `mult_behav_2` and a 100% error rate for the linear and tree multipliers.

Problem 1: Synthesize `mult_behav_1` and `mult_behav_2` following the steps for synthesis on the course procedures page.

(*a*) Indicate the area and critical path delay for each module.

(*b*) Explain why one might be better than the other.

Problem 2: Complete `mult_linear` to that it performs a multiplication using `wid` instances of `good_adder` connected linearly. This module will be sort of a structural version of `mult_behav_2`. Use generate statements to instantiate the adders and make sure that the design is synthesiable.

Note that in this multiplier instance i of the adder cannot start until $i-1$ finishes (that's an oversimplification, but it's true enough).

Problem 3: Complete `mult_tree` so that the adders are connected in a tree-like fashion. Let a and b be the two w-bit operands of the multiplier. There should be $w/2$ adders near the leaves which add two partial products. (There are w partial products, partial product $i \in [0, w-1]$ is $a2^i$ if b_i is 1, or 0 if b_i is 0, where b_i is the digit at bit position i.) At the next level there will be $w/4$ adders which each add the sum of two adders from the lower level, and so on.

First try to solve this using $2w$-bit adders. If you are feeling clever optimize your solution by using $(w+2)$-bit adders for the first row, $(w+4)$-bit adders for the second row, etc.

As before, the design must be synthesiable.

Problem 4: Perform synthesis on your two modules.

(*a*) Indicate the area and delay of each module.

(*b*) Indicate which module you *expected* to be fastest and explain why. If that's different than the one that really is fastest, give a possible reason.

LSU LSU EE 4755 Homework 3 Due: 24 October 2014

Updated 18 October 2014, 18:00:29 CDT

The Homework 3 code package contains a simple behavioral multiplier and several sequential multipliers. It also contains a synthesis script in file `syn.cmd`.

Problem 0: Copy the code package from `/home/faculty/koppel/pub/ee4755/hw/2014f/hw03`. Verify that everything is working by running the simulation on the unmodified file. It should report a 0% error rate for all modules.

Problem 1: The module `mult_seq_csa` is a sequential multiplier that instantiates an adder, however unlike `mult_seq_ga` shown in class, `mult_seq_csa` instantiates a carry-save adder from the Chipware library, `CW_csa`. The carry save adder computes the sum of three integers, `a`, `b`, and `c` (those are the port names). It produces two sums, which we'll call `sum_a` and `sum_b` (the port names for these are `carry` and `sum`). All of these ports are w bits wide, where w is a parameter. The actual sum of `a`, `b`, and `c` is obtained by adding together outputs `sum_a` and `sum_b` using a conventional adder. Carry save adders are used when there many integers to be added. Some arrangement (linear, tree) of many carry-save adders will produce a `sum_a` and `sum_b`, which will be added by a single conventional (called carry-propagate) adder.

The advantage of a carry save adder is that it can compute a sum of w-bit numbers in $O(1)$ time (the amount of time is not affected by w), which of course is much better than the $O(w)$ time for a ripple adder or the $O(\log w)$ time for much more expensive carry look-ahead adders. The performance advantage of a CSA is lost for `mult_seq_csa` because the module only computes one partial product at a time.

(*a*) Sketch the hardware that will be synthesized for `mult_seq_csa`. Show the carry-save adder and other major units as boxes, but be sure to show registers, multiplexors, and other such components. **Do not** show the actual output produced by an actual synthesis program. (It's okay if you look at a synthesis program's output.)

(*b*) Based on this sketch of synthesized hardware, explain why the benefit of using a CSA is lost. Also explain how the module can be made a little faster (with a small change), but is still not a good way to use a CSA.

Problem 2: Module `mult_seq_csa_m` initially contains the m-partial-products-per-cycle module that we did in class. In this problem modify it to use CSA's, and avoid the issue identified in the previous problem.

(*a*) Modify `mult_seq_csa_m` so that it uses the carry-save adder to compute m partial products per cycle. Use `generate` statements to instantiate the CSA's, and of course, connect them appropriately. (In class we used generate statements for the pipelined adder to instantiate stages, that code is in `mult_pipe_ia` in the same file as the assignment.)

(*b*) Sketch the hardware that you expect to be synthesized for an $m = 2$ version. Make sure that your design does not do something foolish with the conventional adder.

Problem 3: Run the synthesis program to compare the cost and performance of `mult_seq_csa_m` to `mult_seq_m`. The synthesis script `syn.cmd` can be used to synthesize these modules at different sizes. To run it use the command `rc -files syn.cmd`. Feel free to modify the script. (It is written in TCL, it should be easy to find information on this language.)

(*a*) Show the cost and performance versus m for these modules.

(*b*) If you solved the previous problem correctly the total delay shown for `mult_seq_csa_m` should be wrong. Explain why, and (optional) if you like try modifying syn.cmd to fix it.

(*c*) Explain how you might expect the delay of `mult_seq_csa_m` to change with increasing m? Explain your expectation and whether the synthesis results bear that out.

LSU EE 4755 Homework 4 Due: 24 November 2014

Problem 0: Copy the code package from `/home/faculty/koppel/pub/ee4755/hw/2014f/hw04`. Verify that everything is working by running the simulation on the unmodified file. It should report that there is correct output but no compression:

```
Correct output, strings match. But no compression!
In size         117 bytes, out size         117 bytes.
```

Problem 1: Module `asc_to_bin` is to filter a stream of ASCII characters so that ASCII decimal numbers are replaced by binary numbers preceded by an escape character. The idea is to reduce the size of data streams that contain lots of large numbers. For example, consider the sentence, "There are 31536000 seconds in a year." The module `asc_to_bin` should replace that sequence of eight ASCII characters 31536000 with an escape character and an integer encoding of the number.

The module has an 8-bit input and output for the character, `char_in` and `char_out`. There is a 1-bit input `can_insert` which is true when the module can read a character from `char_in`. If input `insert_req` is asserted when `can_insert` is true then the character on `char_in` will be read.

There is a 1-bit output `can_remove` which is true when the character on `char_out` is valid. (It would not be valid if the module does not contain any characters and for other reasons.) If input `remove_req` is set to `1` and `can_remove` is true then the character at `char_out` will change to the next character or, if that's the last available character, `can_remove` will go to zero.

There is also a 1-bit input `reset`. If `reset` is high at the positive edge of the clock then the module should reset itself.

Initially in the homework package, module `asc_to_bin` passes through characters unchanged. Modify it so that it converts ASCII decimal numbers to binary as described above.

At the end of the simulation the testbench will indicate whether the output string is correct, and the original and compressed sizes. For example, the output using the unmodified code package will be:

```
Correct output, strings match. But no compression!
In size         117 bytes, out size         117 bytes.
```

The testbench also provides a trace showing some information each time a character is removed. For the unmodified code,

```
ncsim> run
c  79 = O   tail  1 head  0
c 110 = n   tail  3 head  1
c 101 = e   tail  4 head  2
c  32 =     tail  7 head  3
c  49 = 1   tail  8 head  4
```

The character removed is shown as a decimal number and as a character, for example 110 and "n" for the second line. Also shown are the values of two objects in the `asc_to_int` module, `tail` and `head`. Feel free to add your own variables to the list. Search for "Trace execution" to find the code that prints this trace.

The parameter `max\_chars` indicates the maximum size of the integer that should be created. Currently the testbench expects all integers to be of this size.

Keep the following in mind:

- Do not convert a number to binary if it would take more space than the original.
- The module must be synthesizable.
- The synthesized hardware must be reasonably efficient.

For extra credit, modify both the `asc_to_bin` module and the testbench so that `asc_to_bin` can compress a string of ASCII digits to the smallest integer (in multiple of bytes) that can hold the integer. (The current behavior is to use one size integer, determined by parameter `max_chars`.)

Problem 2: Synthesize your module.

(*a*) Indicate the cost and performance with and without timing optimization. (With timing optimization means using `define_clock`.)

(*b*) Even if `define_clock` is used, the synthesis program won't optimize all paths, only those with both ends affected by the clock. Show how to use the Encounter `external_delay` command to get the proper timing optimization.

12 Spring 2001

EE 4702 Homework 1 Due: 12 February 2001

Solve this problem by modifying a copy of `http://www.ee.lsu.edu/v/2001/hw01.html` *which can be found in* `/home/classes/ee4702/files/v/hw01.v`. *See* `http://www.ee.lsu.edu/v/proc.html` *for instructions on running the simulator. Alternate instructions can be found in Lesson 7 of the ModelSim Tutorial, linked to the references web page,* `http://www.ee.lsu.edu/v/ref.html`. *The links are clickable when this assignment is viewed with Acrobat Reader. The ModelSim tutorial and other documentation can also be accessed from the Help menu on the ModelSim GUI (started by the command vsim -gui).*

Problem 1: Copy the homework template, `/home/classes/ee4702/files/v/hw01.v`, into a subdirectory named `hw` in your class account. Simulate the welcome module in the homework template. If it works, a message should tell you to proceed to problem 2.

Problem 2: In Homework 2 (yes, this is Homework 1) a priority encoder will be designed which has an n-bit input and an n-bit output. Let bit positions be numbered from $n-1$ to 0 and let bit zero be the least significant and the rightmost bit when written. Output bit i, $n-1 \geq i \geq 1$, shall be `1` if input bit i is `1` and if input bits $i-1, \ldots, 0$ are all `0`, otherwise output bit i is zero. Output bit 0 is `1` if input bit 0 is `1`, otherwise it is `0`. Therefore, at most one output bit is 1, corresponding to the first input bit that is `1`. Some examples: 0011 → 0001, 0110 → 0010, 0111 → 0001, and 0000 → 0000, where foo → bar indicates that output bar is expected for input foo.

The encoder will be constructed from n cells in the same way a ripple adder is constructed from binary full adder cells. These cells will be designed here, in Homework 1.

Complete module `priority_encoder_1_es` in the homework template so that it is a Verilog explicit structural description of the priority encoder cell. **Do not** rename the module or change any of its ports.

Problem 3: Complete module `priority_encoder_1_is` in the homework template so that it is a Verilog implicit structural description of the priority encoder cell. Do not rename the module or change any of its ports.

Problem 4: Complete module `priority_encoder_1_b` in the homework template so that it is a Verilog behavioral description of the priority encoder cell. Do not rename the module or change any of its ports.

Problem 5: Complete module `test_pe` in the homework template so that it tests the three modules designed above. The test should be by exhaustion. That is, apply all possible combinations of inputs to each module and verify the outputs. (Consider only `0` and `1` for inputs, but watch for `x` or `z` at the outputs, which would indicate an error.)

Module `test_pe` has four one-bit outputs. Output `done` should be set to `1` when the tests are complete. When `done` is `1` outputs `okay_b`, `okay_is`, and `okay_es` shall be set to `1` if the respective module works correctly or set to `0` if the respective module does not work correctly.

As before, do not rename the module or change any of its ports.

EE 4702 Homework 2 Due: 22 Feb 2001, 23:59:59 CST

Solve this problem by modifying a copy of `http://www.ee.lsu.edu/v/2001/hw02.html` *(or .v) which can also be found in* `/home/classes/ee4702/files/v/hw02.v`. *See* `http://www.ee.lsu.edu/v/proc.html` *for instructions on running the simulator. Alternate instructions can be found in Lesson 7 of the ModelSim Tutorial, linked to the references web page,* `http://www.ee.lsu.edu/v/ref.html`. *The links are clickable when this assignment is viewed with Acrobat Reader. The ModelSim tutorial and other documentation can also be accessed from the Help menu on the ModelSim GUI (started by the command vsim -gui).*

In this assignment a priority encoder will be designed which has an n-bit input, called `request`, *and an n-bit output, called* `grant`. *Let bit positions be numbered from* $n-1$ *to* 0 *and let bit zero be the least significant and the rightmost bit when written. Output bit* i, $n-1 \geq i \geq 1$, *shall be* `1` *if input bit* i *is* `1` *and no lower-order bit, if any, is 1. Otherwise output bit* i *is zero. Therefore, at most one output bit is 1, corresponding to the first input bit that is* `1`. *Some examples:* $0011 \rightarrow 0001$, $0110 \rightarrow 0010$, $0111 \rightarrow 0001$, *and* $0000 \rightarrow 0000$, *where foo* $\rightarrow$ *bar indicates that output bar is expected for input foo.*

Problem 1: Complete the module `priority_encoder_8_b` so that it is a behavioral description of an 8-bit priority encoder as described above (and in Homework 1). Just include behavioral code, **do not** instantiate other modules.

Problem 2: Modify the `priority_encoder_1_es` modules from Homework 1 so that each gate has a delay of one cycle.

Problem 3: Complete the module `priority_encoder_8_es` so that it is an explicit structural description of a priority encoder constructed using `priority_encoder_1_es` modules from the previous problem. They may be instantiated within `priority_encoder_8_es` or you can provide an intermediate module, say `priority_encoder_4_es`, which instantiates `priority_encoder_1_es`.

Problem 4: Complete the module `test_pe_8` to that it tests `priority_encoder_8_b` and `priority_encoder_8_es`. Unlike the testbench in Homework 1, this testbench can be commanded to perform the test any number of times. Module `test_pe_8` has one input, `start`, and three outputs, `done`, `okay_b`, and `okay_es`. Initially, `done` should be 0. When `start` is 1 output `done` should be set to zero. At this point, no other outputs should change until `start` goes to zero. After start goes to zero the modules should be tested. When the tests are complete set `okay_b` and `okay_es` based on the outcome of the test. *After* setting `okay_b` and `okay_es` set `done` to 1. At this point, wait for `start` to go to 1 and repeat the process.

Use module `tests_pe_8` (two esses) to test the timing of your testbench. The testbench should be able to catch all errors, including undefined outputs.

EE 4702 Homework 3 Due: 4 April 2001.

Solve this problem by modifying a copy of `http://www.ee.lsu.edu/v/2001/hw03.html` *(or .v) which can also be found in* `/home/classes/ee4702/files/v/hw03.v`. *See* `http://www.ee.lsu.edu/v/proc.html` *for instructions on running the simulator. Alternate instructions can be found in Lesson 7 of the ModelSim Tutorial, linked to the references web page,* `http://www.ee.lsu.edu/v/ref.html`. *The links are clickable when this assignment is viewed with Acrobat Reader. The ModelSim tutorial and other documentation can also be accessed from the Help menu on the ModelSim GUI (started by the command vsim -gui).*

Problem 1: Write a Verilog behavioral description of a microwave oven controller in module `microwave_oven_controller`. The module has two inputs, `key_code` and `clk`. The user operates the oven through a keypad, the keypad has a six-bit output which is connected to the controller through the port named `key_code`. Values for `key_code` are given in the template. As with the calculator described in class, when no key is pressed `key_code` is `key_none` (see the template). The keypad is de-bounced and a user must release one key before pressing another. One-bit input `clk` is a 64 Hz clock.

The controller has six outputs, `beep`, `dmt dmu`, `dst`, `dsu`, and `mag_on`. When one-bit output `beep` is 1 the oven will emit a tone. Four-bit output `dmt` (display minute tens) is connected to the tens digit of the oven minutes display, output `dmu` (display minute units) is connected to the unit digit of the minutes display, `dst` is connected to the tens digit of the seconds display, and `dsu` is connected to the unit digit of the seconds display. The display will properly render digit values 0-9 and will display nothing for a digit value of 10.

When controller output `mag_on` is 1 the magnetron is on (and so the oven is heating).

The keypad has keys for each digit `key_0` - `key_9`, and keys `key_power`, `key_start`, and `key_reset`. There is no popcorn button. The oven operates as follows: When the oven is plugged in it should be placed in a reset state in which the magnetron is off and the display shows zero minutes and zero seconds. To cook at full power, the user enters 1 to 4 digits and presses start. (The digits indicate the cooking time in minutes and seconds. The number of seconds entered must be in $[0, 59]$, so if the user wants to cook for 90 seconds 130 must be entered, not 90.) To cook at some other power the user presses a digit, power, then 1 to 4 digits for the time, then start. Digit 9 indicates 90% of full power, 8 indicates 80% of full power, etc.

Once commanded to start, the oven turns the magnetron on and off until the set time has elapsed. For full power the magnetron stays on over the entire interval. To cook at partial power the magnetron is turned on for a part of each 2.5-second interval. For example, to cook at 30% power the magnetron would be on for 0.75 seconds, off for 1.75 seconds, on for 0.75 seconds, and so on.

The controller must update the display as the user is entering the power and time and while the oven is heating.

If the user presses reset once while the oven is heating the magnetron is turned off but the display should show the remaining time. If the user presses reset again the oven should reset, if the user presses start cooking should resume.

If reset is pressed when the oven is not heating then it will go to the reset state and so any partially entered time or power will be lost.

When cooking is complete the oven should go into the reset state and sound a 2-second beep.

Whenever an invalid key is pressed, even when heating, the oven should emit a 250 ms beep. A key is invalid if it has no meaning when pressed, for example, pressing a digit while heating or pressing start with more than 59 seconds.

Resist the urge to gold plate your submission, for example, by adding outputs for a power indicator or using the display for a clock when not cooking. This will only confuse the TA-bot. Instead, discuss any such ideas with the instructor.

EE 4702 Homework 4 Due: 20 April 2001.

Solve this problem by modifying a copy of `http://www.ee.lsu.edu/v/2001/hw04.html` *(or .v) which can also be found in* `/home/classes/ee4702/files/v/hw04.v`. *See* `http://www.ee.lsu.edu/v/proc.html` *for instructions on running the simulator. Alternate instructions can be found in Lesson 7 of the ModelSim Tutorial, linked to the references web page,* `http://www.ee.lsu.edu/v/ref.html`. *This page also has links to manuals for the synthesis program, Leonardo. The links are clickable when this assignment is viewed with Acrobat Reader. The ModelSim tutorial and other documentation can also be accessed from the Help menu on the ModelSim GUI (started by the command vsim -gui).*

Problem 1: Write a synthesizable Verilog behavioral description of a microwave oven controller in module `microwave_oven_controller` that passes the testbench in `test_oven`. The module is the same as the one assigned in Homework 3 with the following differences. There is a third input, `reset`. The oven should reset if `reset` is one at a positive edge of input `clk`. This is to be used for a power-on reset, it is not the front-panel reset button, and so the oven should reset regardless of what it is doing.

Input `key_code` should only be examined at positive `clk` edges. Input `key_code` will be set to a key's code as long as a key is pressed. Do not expect users to hold down keys for only $\frac{1}{64}$ of a second. As before `key_code` will be `key_none` when no key is pressed.

The module must be synthesizable using the provided synthesis script (see below) and the synthesized hardware must pass the testbench.

Follow these steps:

(1) Write an oven module that passes the testbench (without synthesis). This can be based on your submission to Homework 3, a classmate's submission to Homework 3, or the solution to Homework 3 (when that is posted). Note that the testbench tests the module needed for this homework, which is slightly different than the one designed for Homework 3.

(2) Synthesize the module. This can be done in three ways:

- In Emacs: press `S-F9` (shift f9) while a buffer with the oven module is active. Lines containing error, warning, and information messages will be highlighted. If mouse-2 (the middle button) is pressed while the pointer is over a highlighted message Emacs will jump to the corresponding line in the Verilog description.
- From a shell: type `syn.tcl hw04sol.v`.
- Using the GUI: start Leonardo by selecting "Leonardo" from the slide-up menu over the Emacs kitchen-sink icon on the CDE control panel at the bottom of the screen. Select the SCL05u technology target, under ASIC and Sample. Load the homework solution and press Run Flow. Additional steps are needed to generate Verilog output. Use the first two methods when Verilog output is needed. (The GUI can be used, but the scripts are easier.)

Make sure the module synthesizes (look for a "Synthesis Complete" message), correct any problems if it does not.

(3) Run the testbench on the synthesized module. To do this, load or restart the testbench into Modelsim **without** recompiling it. (The synthesis script should have compiled the synthesized module for you.) If this is done correctly Modelsim should print many lines that look like "Loading work.OR4T2," the names of the technology modules. Run the testbench and correct any errors.

EE 4702 Homework 5 Due: 2 May 2001

Solve this problem by modifying a copy of `http://www.ee.lsu.edu/v/2001/hw05.html` *(or .v) which can also be found in* `/home/classes/ee4702/files/v/hw05.v`. *See* `http://www.ee.lsu.edu/v/proc.html` *for instructions on running the simulator. Alternate instructions can be found in Lesson 7 of the ModelSim Tutorial, linked to the references web page,* `http://www.ee.lsu.edu/v/ref.html`. *This page also has links to manuals for the synthesis program, Leonardo. The links are clickable when this assignment is viewed with Acrobat Reader. The ModelSim tutorial and other documentation can also be accessed from the Help menu on the ModelSim GUI (started by the command vsim -gui).*

Module `bsearch`, in the homework template, stores numbers and can find whether a number had been seen before. The module has four inputs and an output. Input `clk` is a clock, input `reset` is a reset signal, `op` is a command, and `din` in the number to store or find. The module checks commands on a positive edge of the clock, unlike the calculator a command should be present for just one positive edge. (If it is present for two consecutive positive edges it may be performed twice.) Output `result` should be set by the negative edge following the command (though it can be set right after the positive edge). If the result is ready then `result` is set to the appropriate code, explained below, otherwise it is set to `re_busy` until the result is available. The module recognizes three commands, plus a nop.

When `op = op_insert` the module will attempt to store the number present at input `din`, if successful `result` will be set to `re_i_inserted`. If the module were full `result` is set to `re_i_full`. An inserted number must be strictly greater than the last one inserted, if not `result` is set to `re_i_misordered`. When `op = op_find` the module will set `result` to `re_i_present` if the number at `din` was inserted since the last reset, otherwise it is set to `re_i_absent`. When `op = op_reset` the module is emptied.

The homework template contains four copies of a behavioral description of this module, all named `bsearch` and each bracketed by an `‘ifdef`/`‘endif` pair.

The module just below `‘ifdef NOT_SYN` is complete and does not have to be modified. (If would have been Problem 1 if there were more time left in the semester. `:-)`). The other `bsearch` modules are to be converted into synthesizable form as explained in the problems below.

Problem 1: Convert the module below `‘ifdef FORM2` to a synthesizable module in Form 2 *that does one iteration of the forever loop per cycle*. (The original code does the entire loop in one cycle.) The synthesized module must pass the testbench. In the appropriate place in the comments indicate the clock frequency, area (number of gates), and worst-case time needed to find a number (time from positive edge when `op = op_find` to when `result` is set to `re_f_present`).

Problem 2: Convert the module below `‘ifdef FORM3` to a synthesizable module in Form 3 *that does no more than one iteration of the forever loop per cycle*. The synthesized module must pass the testbench. Show how the critical path (as identified by the synthesis program) can be shortened by adding an event control `@(posedge clk)`. Include the line if that would improve performance, otherwise, include it and comment it out. (Remember that performance is more than just clock frequency.) Next to the line indicate the endpoints of the critical path that is, or would be, shortened.

In the appropriate place in the comments indicate the clock frequency, number of gates, and worst-case time needed to find a number.

Problem 3: Convert the module below `‘ifdef FORM3_FAST` to a synthesizable module in Form 3. The synthesized module must pass the testbench. Modify the description so that two iterations

of the original code is done by one iteration (and clock cycle) in the modified code. This module should take fewer clock cycles than the one in the previous problem (nearly half when the capacity is large). In the appropriate place in the comments indicate the clock frequency, number of gates, and worst-case time needed to find a number.

The modules must be synthesizable using the provided synthesis script (see below) and the synthesized hardware must pass the testbench.

Follow these steps:

(1) Modify the modules as needed. Be sure to include a `` `define FOO `` when you are working on a module next to `` `ifdef FOO ``.

(2) Synthesize the module. This can be done in two ways:

- In Emacs: press `S-F9` (shift f9) while a buffer with the Verilog description is active. Lines containing error, warning, and information messages will be highlighted. If mouse-2 (the middle button) is pressed while the pointer is over a highlighted message Emacs will jump to the corresponding line in the Verilog description.
- From a shell: type `syn.pl hw05sol.v`.

The clock frequency, number of gates, and critical path information are written by the synthesis program and script.

Make sure the module synthesizes (look for a "Synthesis Complete" message), correct any problems if it does not.

(3) Run the testbench on the synthesized module. To do this, load or restart the testbench into Modelsim **without** recompiling it. (The synthesis script should have compiled the synthesized module for you.) If this is done correctly Modelsim should print many lines that look like "Loading work.OR4T2," the names of the technology modules. Run the testbench and correct any errors.

13 Spring 2000

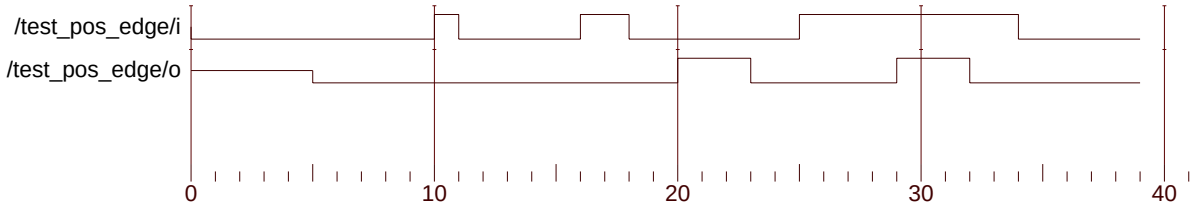
EE 4702 Homework 1 Due: 14 February 2000

Solve this problem by modifying a copy of `http://www.ee.lsu.edu/v/2000/hw01.v`. *Use Lesson 7 of the ModelSim tutorial for instructions on using the simulator as described in the references web page, http://www.ee.lsu.edu/v/ref.html. Instructions for submitting a solution will be given later.*

Problem 1: Write two Verilog descriptions of the following circuit. The circuit has a four-bit input on which integers will appear. If the integer is equal to 2 or 9 the output should be 1, otherwise the output should be zero. One description, in a module named `number_detect_es`, should be explicit structural, and the other should be implicit structural in a module named `number_detect_is`.

Problem 2: Write a testbench for the descriptions above. Test all possible inputs. Name the testbench module `test_number_detect`.

Problem 3: The structural module below, when finished, is to produce a pulse of duration 3 ns on output `o` starting 4 ns after a positive edge on input `i`, but only if `i` is 1 for at least 2 ns. (The finished module will remain structural.) Correct operation is shown in the sample timing below where there are three pulses on input `i`. No output pulse appears at 14 ns because the input is `1` for only 1 ns. Pulses on `o` are produced for the next two positive edges on `i`. The testbench code used to generate the waveforms is in module `test_pos_edge`, already written.



Entity:test_pos_edge Architecture: Date: Wed Feb 02 17:54:43 CST 2000 Row: 1 Page: 1

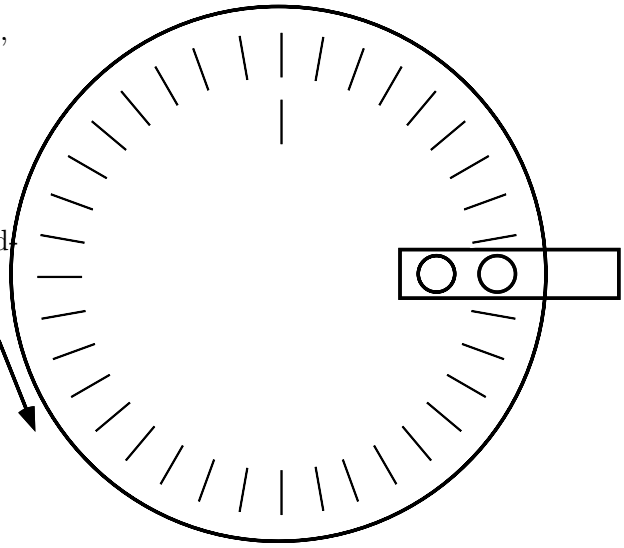
```
module pos_edge_trigger(o,i);
   input i;
   output o;
   wire   noti;
   wire   preout;

   assign o = preout;

   not (noti,i);
   and (preout,i,noti);

endmodule // pos_edge_trigger
```

Add delay specifications so that the module works as described. Add **only** delay specifications, nothing else. Don't add gates, don't add modules, and especially don't add behavioral code.

EE 4702 Homework 2 Due: 23 February 2000

Homework 2 and 3 are being assigned simultaneously. Homework 3 is really just homework 2a, but calling it that would ruin the numbering scheme. Solution templates can be found in `/home/classes/ee4702/files/v` *and will be linked to the web page. Instructions for submission will be posted later.*

Problem 1: A tachometer measures rotation rate by detecting marks on a disk using photodetectors as illustrated below.



In the illustration there are two rings of marks, in this assignment only the outer ring (the one with lots of marks) will be used.

As the disk spins the number of marks passing under the disk are counted. At fixed intervals a rotation rate is updated.

Write a Verilog behavioral description for hardware that determines the rotation rate using the photodetector output. The module has the following declaration:

```
module tach1(rpx,pd,clk);
   input pd, clk;
   output rpx;
   wire   pd, clk;
   reg [9:0] rpx;

   parameter  freq = 500;    // Clock frequency.
   parameter  marks = 4;     // Number of marks on ring.
   parameter  update_interval = 0.5;  // Update every update_interval seconds.
   parameter  perwhat = 60;  // Measure in revolutions per 60 seconds.

// Solution goes here.

endmodule
```

Input `clk` is a square wave for use by the module. Input `pd` is the photodetector output. It is `1` when a mark is under the photodetector. Output `rpx` is the rotation rate. Parameter `freq` is the frequency of `clk` and `marks` is the number of marks on a disk. Parameter `update_interval` is the number of seconds between updates of `rpx`. For example, if `update_interval` were 3 then `rpx` would have to be updated every 3 seconds. Parameter `perwhat` is the time unit for measuring revolutions, in seconds. If it is `60` then `rpx` should be in revolutions per minute, if it is `1` then `rpx` should be in revolutions per second, etc.

Consider the instantiation below:

```
   tach1  #(200) s1(rpx,pd,clk);
```

This instantiates a tachometer which is to use a 200 MHz clock.

Call $\frac{\texttt{perwhat}}{\texttt{marks}\times\texttt{update\_interval}}$ the *precision*, p. Let n_m denote the number of marks that have been counted in a time interval of duration `update_interval`. Then `rpx` should be set to $n_m \times p$.

In addition to generating `rpx` the module should also check to make sure its parameters are suitable. The parameters are not suitable if the precision is not an integer or if any registers would overflow in normal operation.

Use the testbenches provided in the solution template to test your circuit. Testbench module `test_tach1_fast` tests a single instance, while `test_tach1_detailed` tests several instances (using different parameters).

Follow the following rules when writing the hardware description. (The rules do not apply to testbench code.)

- Do not use multipliers or dividers.
- Do not use delays: `#3 i=1;`. You can use event controls: `@(posedge clk)`.
- Use the initial block for parameter verification and register initialization only.

To be continued in homework 3 . . .

EE 4702 Homework 3 Due: 3 March 2000

Homework 3 is being split in to homework 3 and 4. Homework 4 is really just homework 2b, but calling it that would ruin the numbering scheme. Solution templates can be found in `/home/classes/ee4702/files/v` *and will be linked to the web page. Instructions for submission can be found in the homework template.*

Problem 1: Write an implicit structural description of the module designed in homework 2, either your design or the posted solution. Note: see the template for a workaround to a bug when using parameters.

Problem 2: Design a behavioral description of hardware similar to the one from homework 2, but that measures rotation speed by measuring the time between marks. The inputs and output are the same and the parameters are the same, except that `update_interval` is missing. In this module the output should be updated for each detected mark. (The update does not have to occur on the positive edge of `pd`, but it does have to be updated sometime.) The output must correctly indicate zero rotation rate. (You'll see why that needed to be specified.) Though the number of marks is known the width of the marks is not.

```
module tach2(rpx,pd,clk);
   input pd, clk;
   output rpx;
   wire   pd, clk;
   reg [9:0] rpx;

   parameter  freq = 500;    // Clock frequency.
   parameter  marks = 4;     // Four pulses per revolution.
   parameter  perwhat = 60;  // Measure in revolutions per 60 seconds.
// Code here.

endmodule
```

Follow the following rules when writing the hardware description. (The rules do not apply to testbench code.)

- You can use multipliers or dividers. (Just use the usual operators, no need to instantiate anything.)
- Do not use delays: `#3 i=1;`. You can use event controls: `@(posedge clk)`.
- Use the initial block for parameter verification and register initialization only.

EE 4702 Homework 4 Due: 17 March 2000

Homework 4 is really just homework 2b, but calling it that would ruin the numbering scheme. Solution templates can be found in `/home/classes/ee4702/files/v` *and will be linked to the web page.*

Changes made to this assignment 13 March 2000, 10:02:54 CST. Changes are shown in a slanted (not italic) font.

Problem 1: Suppose the marks are glued on the disks used in the problems above and that sometimes they fall off. (Or maybe they're stolen, or painted over.) Design a behavioral Verilog module that can compute the correct rotation rate when as few as $\lceil \frac{m+1}{2} \rceil$ marks are still present, where m is the original number of marks. The angle subtended by the marks (their width, sort of) is not known.

Use the same design rules as for `tach2`. You may base the solution to this problem on your solution to homework 3 (perhaps corrected) or the posted solution to homework 3.

The module does **not** *have to measure zero correctly, when the rotation rate is below the minimum measurable speed any output is acceptable.*

```
module tach3(rpx,pd,clk);
   input pd, clk;
   output rpx;
   wire   pd, clk;
   reg [9:0] rpx;

   parameter  freq = 500;    // Clock frequency.
   parameter  marks = 12;    // Four pulses per revolution, when new.
   parameter  perwhat = 60;  // Measure in revolutions per 60 seconds.
// Code here.

endmodule
```

Problem 2: Design a testbench for the code above. The testbench should test the ability of `tach3` to work with missing marks. The testbench can be based on the `tach2` testbench provided with homework 3.

The testbench should be able to handle a disk with up to one hundred marks. Test at least these patterns: all marks present, one mark missing, the maximum number of marks missing and spread out as much as possible (so almost every other mark is missing), and the maximum number of marks missing where the missing marks are all adjacent (so there will be a big gap). Also, add a pattern of your own.

See the hint at `http://www.ee.lsu.edu/v/2000/hw04hint.html`.

EE 4702 Homework 5 Due: 19 April 2000

Solution templates can be found in `/home/classes/ee4702/files/v` *and will be linked to the web page. Put your solution in a file named hw05sol.v. Soon after the time the assignment is due your directory tree will be searched for files named hw05sol.v and the most-recently modified one will be copied. If no such file is found an attempt will be made to copy a file using a guessed name, but this is not something to be relied on. Give the file the correct name.*

Solutions to the problems below should be synthesized for the following technology: ASIC (type of target), Sample (manufacturer [usually]), XCL05U (technology family). Do not specify any optimization or other synthesis options. View the RTL schematic to check your solutions. (Under the Tools menu or using the toolbar button.) Leonardo is started by typing `leonardo &` *in a shell. To work around a cosmetic stdout bug start Leonardo by typing* `leonardo > /dev/null &` *. Additional instructions on running Leonardo will be posted later.*

The assignments will be graded under the assumption that the schematic was viewed; a substantial number of points will be deducted for solutions that do not synthesize correctly.

Problem 1: Complete the Leonardo-synthesizable Verilog description of an ALU module shown below. The module has three inputs, `a`, `b`, and `op`. Inputs `a` and `b` are each 8 bits and hold unsigned integers. Input `op` specifies an operation to perform; the coding is given by the parameters. The module has two outputs, `res` and `err`. Output `res` is 8 bits and is the result of performing the operation; output `err` is one bit and is 1 if `res` cannot hold the result of `op`. That is, `err` is one if the sum is more than eight bits or the difference is negative; it is zero otherwise.

Write the description using behavioral code and synthesize it for the target specified above. The synthesized module should be combinational—no latches allowed. The module should perform only the three operations indicated, **don't** add your own.

```
module alu(res,err,a,b,op);
   input a, b, op;
   output res, err;

   parameter op_add = 0, // Addition.
             op_sub = 1, // Subtraction
             op_and = 2; // Bitwise and.

   // Insert solution here.  It's okay to delete this comment.

endmodule // alu
```

Problem 2: Complete the design of a Leonardo-synthesizable Verilog module with four 1-bit outputs and six one-bit inputs with the following behavior when synthesized:

- Output `w` is equal to the value input `d` had at the last negative edge of `clk`. (In other words, `w` is set to `d` at the negative edge of `clk`.)
- Output `y` is equal to the value input `a` had at the last positive edge of `clk`.
- Output `z` is equal to the last value input `c` had when both `clk` was high and `d` = `b`.
- Output `x` set to `b` at positive edge of `clk` if `a`=1. Output `x` is set to 1 if `clk`=1 and `d`=`a`.
- All outputs are set to zero when input `r`=1 (and will remain zero until set to a new value as described above).

If might be helpful to figure out what kinds of flip-flops are needed for each output and then check if Leonardo chooses the correct one (or something equivalent).

```
module latch_thing(w,x,y,z,a,b,c,d,r,clk);
   input a, b, c, d, r, clk;
   output w, x, y, z;

   // Insert solution here.  It's okay to delete this comment.

endmodule // latch_thing
```

EE 4702 Homework 6 Due: 28 April 2000

Solution templates can be found in `/home/classes/ee4702/files/v` *and* `www.ee.lsu.edu/v/2000/hw06.html` *Put your solution in a file named hw06sol.v. Soon after the time the assignment is due your directory tree will be searched for files named hw06sol.v and the most-recently modified one will be copied. If no such file is found an attempt will be made to copy a file using a guessed name, but this is not something to be relied on. Give the file the correct name.*

Solutions to the problems below should be synthesized for the following technology: ASIC (type of target), Sample (manufacturer [usually]), XCL05U (technology family). Do not specify any optimization or other synthesis options. View the RTL schematic to check your solutions. (Under the Tools menu or using the toolbar button.) Leonardo is started by typing `leonardo &` *in a shell. To work around a cosmetic stdout bug start Leonardo by typing* `leonardo > /dev/null &` *. See the procedures and FAQ web pages for additional instructions on running Leonardo.*

See the FAQ page for instructions on how to write Verilog code of the synthesized module for simulation using the Leonardo GUI, a TCL script, or Emacs. The process is particularly convenient using Emacs.

The assignments will be graded under the assumption that the synthesized code was simulated using the testbench provided; a substantial number of points will be deducted for solutions that do not pass the testbench correctly.

Problem 1: A Verilog behavioral description of a module similar to the one described in the first midterm problem appears in `http://www.ee.lsu.edu/v/2000/hw06.v`. The module cannot be synthesized by Leonardo (1999.1f). Modify the module so that it can be synthesized while retaining the benefits of behavioral code. (That is, do not convert it to a structural description.) The synthesized code must pass the testbench provided with the code.

The module, `width_change`, describes a FIFO in which data is inserted in 4-bit *nibbles* (the technical term for half a byte, no kidding) and removed in 8-bit bytes. The total storage capacity is 32 bits. In addition to the 4-bit input and 8-bit output there are two 1-bit inputs, `inclk` and `outclk`. On a positive edge of `inclk` data is read; on a positive edge of `outclk` data is removed in FIFO fashion. The low nibble of the output (bits 0-3) holds data that arrived earlier than the high nibble of the output. Output `full` is one if the FIFO cannot accept another nibble, output `empty` is one if the FIFO is empty (in which case the output must be all zeros), and output `complete` is 1 if the output has eight bits of data. (Output `complete` is zero if the FIFO is empty or if it contains just 4 bits. If the FIFO contains four bits the high nibble of output should be zeros.)

Note that, unlike the test question, the sizes of the input, output, and storage capacity are digital-logic-friendly powers of two. However like the midterm exam, the input and output each have their own positive edge triggered clock. Getting this into synthesizable form will take some thought.

14 Fall 2024 Solutions

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2024 Homework 1 -- SOLUTION
//

 /// Assignment https://www.ece.lsu.edu/koppel/v/2024/hw01.pdf

`default_nettype none

////////////////////////////////////////////////////////////////////////////////
/// Problem 1
//
  /// Complete the dot product modules as described below.
//
//     [✔] Complete dot2.
//     [✔] Complete dot3.
//     [✔] Complete dot4.
//     [✔] None of your modules can use assign statements or procedural code.
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//
//     [✔] Don't assume any particular parameter values.
//
//     [✔] Code must be written clearly.


module mult
  #( int w = 5 ) ( output uwire [w-1:0] p, input uwire [w-1:0] a, b );
   // Do not modify this module.
   assign p = a * b;
endmodule

module add
  #( int w = 5 ) ( output uwire [w-1:0] s, input uwire [w-1:0] a, b );
   // Do not modify this module.
   assign s = a + b;
endmodule


module dot2
  #( int w = 5 )
   ( output uwire [w-1:0] dp,
     input uwire [w-1:0] a[1:0], b[1:0] );

   // Compute
   //
   //   dp = a[0] * b[0] + a[1] * b[1];
   //
   // [✔] Use as many instantiations of mult and add as are needed, if any.
   // [✔] Try to minimize the number of instantiations.
   // [✔] DO NOT use assign, and DO NOT use procedural code (always, etc.).
   //
   // It will also be necessary to declare uwire objects to interconnect
   // the instantiations.

   /// SOLUTION
```

```
   uwire [w-1:0] p0, p1;

   mult #(w) m0(p0, a[0], b[0] );
   mult #(w) m1(p1, a[1], b[1] );
   add #(w) ad(dp, p0, p1 );

endmodule

module dot3
  #( int w = 5 )
   ( output uwire [w-1:0] dp,
     input uwire [w-1:0] a[2:0], b[2:0] );

   // Compute
   //
   //   dp = a[0] * b[0] + a[1] * b[1] + a[2] * b[2];
   //
   // [✔] Use at least one instantiation of dot2.
   // [✔] Use as many instantiations of mult and add as are needed, if any.
   // [✔] Try to minimize the number of instantiations.
   // [✔] DO NOT use assign, and DO NOT use procedural code (always, etc.).

   /// SOLUTION
   uwire [w-1:0] p0, p2;

   dot2 #(w) d0( p0, a[1:0], b[1:0] );
   mult #(w) m2( p2, a[2], b[2] );
   add #(w) a2(dp, p0, p2 );

endmodule

module dot4
  #( int w = 5 )
   ( output uwire [w-1:0] dp,
     input uwire [w-1:0] a[3:0], b[3:0] );

   // Compute
   //
   //   dp = a[0] * b[0] + a[1] * b[1] + a[2] * b[2] + a[3] * b[3];
   //
   // [✔] Use as many instantiations of dot2 and dot3 as are needed, if any.
   // [✔] Use as many instantiations of mult and add as are needed, if any.
   // [✔] Try to minimize the number of instantiations.
   // [✔] Try to minimize the time to compute the operation based on the
   //     longest path from inputs to outputs.
   // [✔] DO NOT use assign, and DO NOT use procedural code (always, etc.).

   /// SOLUTION
   uwire [w-1:0] p0, p1;

   dot2 #(w) d0( p0, a[1:0], b[1:0] );
   dot2 #(w) d1( p1, a[3:2], b[3:2] );
   add #(w) ad(dp, p0, p1 );

endmodule



////////////////////////////////////////////////////////////////////////////////
```

```
/// Problem 2
//
  /// Complete the MADD Dot Product Modules
 ///
//
//     [✔] Complete dot2m.
//     [✔] Complete dot4m.
//     [✔] Complete dot6m.
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//
//     [✔] Don't assume any particular parameter values.
//
//     [✔] Code must be written clearly.


module madd
  #( int w = 8 )
   ( output uwire [w-1:0] s,  input uwire [w-1:0] si, a, b );
   // Do not modify this module.
   assign s = si + a * b;  // Perform a Multiply Add Operation.
endmodule

module dot2m
  #( int w = 5 )
   ( output uwire [w-1:0] dp,
     input uwire [w-1:0] si, a[1:0], b[1:0] );

   // Compute
   //
   //   dp = si + a[0] * b[0] + a[1] * b[1];
   //
   // [✔] Use as many instantiations of madd as are needed, if any.
   // [✔] Try to minimize the number of instantiations.
   // [✔] DO NOT use assign, and DO NOT use procedural code (always, etc.).

   /// SOLUTION
   uwire [w-1:0] p0;

   madd #(w) m0(p0, si, a[0], b[0] );
   madd #(w) m1(dp, p0, a[1], b[1] );

endmodule


module dot4m
    #( int w = 6 )
   ( output uwire [w-1:0] dp,
     input uwire [w-1:0] si, a[3:0], b[3:0] );

   // Compute
   //
   //   dp = si + a[0] * b[0] + a[1] * b[1] + a[2] * b[2] + a[3] * b[3];
   //
   // [✔] Use as many instantiations of dot2m and madd as are needed, if any.
   // [✔] Try to minimize the number of instantiations.
   // [✔] DO NOT use assign, and DO NOT use procedural code (always, etc.).
```

```
   /// SOLUTION
   uwire [w-1:0] p0;

   dot2m #(w) d0( p0, si, a[1:0], b[1:0] );
   dot2m #(w) d1( dp, p0, a[3:2], b[3:2] );

endmodule

module dot6m
    #( int w = 7 )
   ( output uwire [w-1:0] dp,
     input uwire [w-1:0] si, a[5:0], b[5:0] );

   // Compute
   //
   //   dp = si+a[0]*b[0]+a[1]*b[1]+a[2]*b[2]+a[3]*b[3]+a[4]*b[4]+a[5]*b[5];
   // Using the minimum number of instantiations of:
   //   dot2m, dot4m, madd.
   //
   // [✔] Use as many instantiations of dot2m,dot4m, and madd as needed, if any.
   // [✔] Try to minimize the number of instantiations.
   // [✔] DO NOT use assign, and DO NOT use procedural code (always, etc.).

   /// SOLUTION
   uwire [w-1:0] p0;

   dot4m #(w) d0( p0, si, a[3:0], b[3:0] );
   dot2m #(w) d1( dp, p0, a[5:4], b[5:4] );

endmodule




////////////////////////////////////////////////////////////////////////////////
///  Problem 3
//
  ///  Complete hybrid sizes
 ///
//
//     [✔] Complete dot2y.
//     [✔] Complete dot4y.
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//
//     [✔] Don't assume any particular parameter values for wa and wb.
//
//     [✔] Code must be written clearly.




module multy
  #( int wa = 1, wb = 2, wo = wa + wb )
   ( output uwire [wo-1:0] p,
     input uwire [wa-1:0] a,
     input uwire [wb-1:0] b );
```

```
    /// SOLUTION
    //
    //  Each port has its own size!

    // [✔] Modify the connections to this module.  (The stuff above this line.)

    assign p = a * b;

endmodule

module addy
  #( int wi = 1, wo = wi + 1 )
   ( output uwire [wo-1:0] s,
     input uwire [wi-1:0] a, b );

    /// SOLUTION
    //
    //  The output can be a different size, wo, than the input, wi.

    // [✔] Modify the connections to this module. (The stuff above this line.)

    assign s = a + b;

endmodule


module dot2y
  #( int wa = 5, wb = 6, wo = wa + wb + 1 )
   ( output uwire [wo-1:0] dp,
     input uwire [wa-1:0] a[1:0],
     input uwire [wb-1:0] b[1:0] );

    // Compute
    //
    //   dp = a[0] * b[0] + a[1] * b[1];
    //
    // Note: Product of a wa-bit and wb-bit unsigned integer needs wa+wb bits.
    //       Sum of a wa-bit and wb-bit integer needs max(wa,wb)+1 bits.
    //
    // [✔] Modify addy and multy so that in can be instantiated ..
    //     .. with ports set to the needed size.
    // [✔] Do not make the ports larger than are needed.
    // [✔] Use as many instantiations of multy and addy as are needed, if any.
    // [✔] DO NOT use assign, and DO NOT use procedural code (always, etc.).

    /// SOLUTION


    // Compute the size needed to hold the product a[i] * b[i].
    //
    localparam int wr = wa + wb;
    //
    // Use that size for the dot product terms.
    //
    uwire [wr-1:0] p0, p1;

    multy #(wa,wb,wr) m0( p0, a[0], b[0] );
    multy #(wa,wb,wr) m1( p1, a[1], b[1] );
    addy #(wr,wr+1) ad( dp, p0, p1 );
```

```
endmodule


module dot4y
  #( int wa = 5, wb = wa, wo = wa + wb + 2)
   ( output uwire [wo-1:0] dp,
     input uwire [wa-1:0] a[3:0],
     input uwire [wb-1:0] b[3:0] );

   // Compute
   //
   //   dp = a[0] * b[0] + a[1] * b[1] + a[2] * b[2] + a[3] * b[3];
   //
   //   addy, multy, dot2y
   // Note: Product of a wa-bit and wb-bit unsigned integer needs wa+wb bits.
   //       Sum of a wa-bit and wb-bit integer needs max(wa,wb)+1 bits.
   //
   // [✔] Use as many instantiations of multy,addy, and dot2y as needed, if any.
   // [✔] DO NOT create a new version of multy or addy just for dot4y.
   // [✔] Do not make the ports larger than are needed.
   // [✔] DO NOT use assign, and DO NOT use procedural code (always, etc.).

   /// SOLUTION
   //
   // Compute the size needed to hold the dot2 results.
   //
   localparam int wr = wa + wb + 1;

   uwire [wr-1:0] p0, p1;

   dot2y #(wa,wb) d0( p0, a[1:0], b[1:0] );
   dot2y #(wa,wb) d1( p1, a[3:2], b[3:2] );
   addy #(wr,wo) ad(dp, p0, p1 );

endmodule



////////////////////////////////////////////////////////////////////////////////
/// Testbench Code
//
// It is okay to modify the testbench code to facilitate the coding
// and debugging of your modules. Keep in mind that your submission
// will be tested using a different testbench, so on the one hand no
// one will be accused of dishonesty for modifying the testbench
// below. However be sure to restore any changes to make sure that
// your code passes the original testbench.


// cadence translate_off


module testbench;

   localparam int npsets = 6; // This MUST be set to the size of pset.
   // { variation wa wb delta }
   localparam int pset[npsets][4] =
              '{
```

```
                { 0, 2, 0, 0 },
                { 0, 8, 0, 0 },
                { 1, 3, 0, 0 },
                { 1, 7, 0, 0 },
                { 2, 5, 3, 0 },
                { 2, 8, 4, 0 }
                };

   logic d[npsets:-1]; // Start / Done signals.

   int t_errs[npsets];
   int t_errs_mod[string];
   int t_n_tests_mod[string];
   int t_n_tests[npsets];
   string sname[] = { "Prob 1", "Prob 2", "Prob 3" };

   initial begin
      for ( int i=0; i<npsets; i++ ) begin
         t_errs[i] = 0;
         t_n_tests[i] = 0;
      end
      d[-1] = 1;

      wait( d[npsets-1] );
      $write("\n");
      for ( int p=0; p<0; p++ ) begin
         automatic string wline =
           pset[p][0] < 2 ? $sformatf("w=%0d", pset[p][1]) :
             $sformatf("wa=%0d,wb=%0d", pset[p][1],pset[p][2]);
        $write("End of tests %s, %s: %0d errors out of %0d tests.\n",
               sname[ pset[p][0] ], wline, t_errs[p], t_n_tests[p]);
      end

      foreach ( t_errs_mod[mname] ) begin
         $write("End of tests. For %s:  %0d errors out of %0d tests.\n",
                mname, t_errs_mod[mname], t_n_tests_mod[mname] );
      end

   end

   for ( genvar p=0; p<npsets; p++ ) begin
      testbench_n #( .idx(p), .variation(pset[p][0]), .wap(pset[p][1]), .wbp(pset[p][2]), .wo_delta(p) )
      tb( .done(d[p]), .tstart(d[p-1]) );
   end

endmodule

module testbench_n
  #( int idx, variation = 0, wap = 2, wbp = 3, wo_delta = 1 )
   ( output logic done, input uwire tstart );

   localparam bit normal = variation == 0;
   localparam bit ripple = variation == 1;
   localparam bit hybrid = variation == 2;

   localparam int wa = wap;
   localparam int wb = hybrid ? wbp : wap;
   localparam int wo = hybrid ? wa + wb + 2 : wap;
```

```
   localparam int n_tests = 1000;
   localparam int ms = 7; // Maximum array size

   logic [wa-1:0] a[ms-1:0];
   logic [wb-1:0] b[ms-1:0];
   uwire [wo-1:0] dp[ms];
   logic [wo-1:0] z;

   int sizes[$];
   string npre, npost;

   localparam int wo2 = wa + wb + 1;

   if ( ripple ) begin
      dot2m #(wa) d2( dp[2], z, a[1:0], b[1:0] );
      dot4m #(wa) d4( dp[4], z, a[3:0], b[3:0] );
      dot6m #(wa) d6( dp[6], z, a[5:0], b[5:0] );
      initial begin sizes = { 2, 4, 6 }; npre = "dot"; npost = "m"; end
   end else if ( normal ) begin
      dot2 #(wa) d2( dp[2], a[1:0], b[1:0] );
      dot3 #(wa) d3( dp[3], a[2:0], b[2:0] );
      dot4 #(wa) d4( dp[4], a[3:0], b[3:0] );
      initial begin sizes = { 2, 3, 4 }; npre = "dot"; npost = ""; end
   end else begin
      uwire [wo2-1:0] dp2o;
      dot2y #(wa,wb) d2( dp2o, a[1:0], b[1:0] );
      assign dp[2] = { 1'b0, dp2o };
      dot4y #(wa,wb) d4( dp[4], a[3:0], b[3:0] );
      initial begin sizes = { 2, 4 }; npre = "dot"; npost = "y"; end
   end

   string names[$];

   initial begin

      automatic int n_err_dp[ms] = '{ms{0}};
      automatic string wline = hybrid
        ? $sformatf("wa=%0d,wb=%0d", wa, wb) : $sformatf("w=%0d", wa);

      wait( tstart );

      $write("Starting tests for %s %s.\n", testbench.sname[variation], wline);

      foreach ( sizes[sidx] )
        names.push_back( $sformatf("%s%0d%s",npre,sizes[sidx],npost) );

      foreach ( sizes[sidx] ) begin

         automatic int sz = sizes[sidx];

         for ( int t=0; t<n_tests; t++ ) begin

            automatic logic [wo-1:0] shadow_dp[ms-1:-1];
            z = {$random};
            shadow_dp[0] = ripple ? z : 0;

            for ( int i=0; i<=sz; i++ ) begin
               automatic int bits = wa + wb + $clog2(i+1);
               // Test Patterns. Okay to Modify.
```

```
            case (t)
              0: begin a[i]=0; b[i]=0; end
              1: begin a[i]=0; b[i]=1; end
              2: begin a[i]=1; b[i]=0; end
              3: begin a[i]=1; b[i]=1; end
              4: begin a[i]=1; b[i]=i; end
              5: begin a[i]=1; b[i]=(1<<wb)-1; end
              6: begin a[i]=(1<<wa)-1; b[i]=1; end
              default: begin a[i] = {$random}; b[i] = {$random}; end
            endcase
            shadow_dp[i+1] = ( shadow_dp[i] + a[i] * b[i] ) % 2**bits;
         end

         #1;

         if ( shadow_dp[sz] !== dp[sz] ) begin

            n_err_dp[sz]++;
            if ( n_err_dp[sz] < 6 ) begin
               $write( "Error, %s, %s, %0d != %0d (correct)\n",
                       names[sidx], wline, dp[sz], shadow_dp[sz] );
               for ( int j=0; j<sz; j++ )
                 $write(" %0d * %0d%s",
                        a[j], b[j], j == sz-1 ? "\n" : " +");
               for ( int j=0; j<sz; j++ )
                 $write(" 0x%0x * 0x%0x%s",
                        a[j], b[j], j == sz-1 ? "\n" : " +");
            end
         end
      end
   end

   testbench.t_errs[idx] = 0;
   $write("Tests complete %s %s: ",
          testbench.sname[variation], wline);

   foreach ( sizes[sidx] ) begin

      automatic int sz = sizes[sidx];
      testbench.t_errs_mod[names[sidx]] += n_err_dp[sz];
      testbench.t_n_tests_mod[names[sidx]] += n_tests;

      testbench.t_errs[idx] += n_err_dp[sz];
      testbench.t_n_tests[idx] += n_tests;
      $write("%s %0d errors, ", names[sidx],n_err_dp[sidx]);
   end
   $write("\n");



   done = 1;
end

endmodule

// cadence translate_on
```

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.

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2024 Homework 2 -- SOLUTION
//
```

/// Assignment https://www.ece.lsu.edu/koppel/v/2024/hw02.pdf
/// Solution writeup https://www.ece.lsu.edu/koppel/v/2024/hw02_sol.pdf

```
`default_nettype none


////////////////////////////////////////////////////////////////////////////////
/// Problem 1
//
```

/// Complete scaled_comp_fp so the comparison is done in FP.

```
//
//     [✔] Use Chipware modules for floating-point operations and conversions.
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//
//     [✔] Don't assume any particular parameter values.
//
//     [✔] Code must be written clearly.

typedef enum logic [2:0]
  { Rnd_to_even = 0, Rnd_to_0 = 1, Rnd_to_plus_inf = 2,
    Rnd_to_minus_inf = 3, Rnd_to_near_up = 4, Rnd_from_0 = 5 }
    Rnd;

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 );

   /// SOLUTION -- Problem 1

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

   // 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;
   //
   // This just appends w_sig2-w_sig zeros to the LSB part.

   // 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.

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Problem 2
//
  /// Complete scaled_comp_int so the comparison is done as an int.
//
//     [✔] Use Chipware modules for floating-point operations and conversions.
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//
//     [✔] Don't assume any particular parameter values.
//
//     [✔] Code must be written clearly.

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 );

    /// SOLUTION -- Problem 2

    // 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
    // will not be 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];
    uwire ssum_overflow = sumi_status[6];

    // Compute gt
```

```
   //
   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



////////////////////////////////////////////////////////////////////////////////
/// Testbench Code


// cadence translate_off

virtual class conv #(int wexp=6, wsig=10);
   // Convert between real and fp types using parameter-provided
   // exponent and significand sizes.

   localparam int w = 1 + wexp + wsig;
   localparam int bias_r = ( 1 << 11 - 1 ) - 1;
   localparam int w_sig_r = 52;
   localparam int w_exp_r = 11;
   localparam int bias_h = ( 1 << wexp - 1 ) - 1;

   static function logic [w-1:0] rtof( real r );
      logic [wsig-1:0] sig_f;
      logic [w_sig_r-wsig-2:0] sig_x;
      logic sig_x_msb;
      logic [w_exp_r-1:0] exp_r;
      logic sign_r;
      { sign_r, exp_r, sig_f, sig_x_msb, sig_x } = $realtobits(r);
      // So, what about a rounding mode? Not now!
      rtof = !r ? 0 : { sign_r, wexp'( exp_r + bias_h - bias_r ), sig_f };
   endfunction

   static function real ftor( logic [w-1:0] f );
      ftor = !f ? 0.0
        : $bitstoreal
          ( { f[w-1],
              w_exp_r'( bias_r + f[w-2:wsig] - bias_h ),
              f[wsig-1:0], (w_sig_r-wsig)'(0) } );
   endfunction

   static function int err_bits( logic [w-1:0] a, b );

      logic [wsig-1:0] sig_a, sig_b;
      logic [wsig+2:0] frac_a, frac_b, frac_diff;
      logic [wexp-1:0] exp_a, exp_b;
      logic s_a, s_b;
      int delta_e;

      if ( $isunknown(a) || $isunknown(b) ) return 1 << wexp;
      if ( a == b ) return 0;

      { s_a, exp_a, sig_a } = a;
      { s_b, exp_b, sig_b } = b;

      if ( exp_a == 0 || exp_b == 0 ) begin
         logic [wsig-1:0] sig = ~ ( sig_a | sig_b );
         return 1 + wsig - $clog2( sig + 1 );
      end
```

```
      delta_e = $abs( 0 + exp_a - exp_b );
      if ( delta_e > 1 ) return delta_e + wsig;
      frac_a = exp_a > exp_b ? { 2'b1, sig_a, 1'b0 } : { 3'b1, sig_a };
      frac_b = exp_b > exp_a ? { 2'b1, sig_b, 1'b0 } : { 3'b1, sig_b };
      frac_diff =
        s_a != s_b ? frac_a + frac_b :
        frac_a > frac_b ? frac_a - frac_b : frac_b - frac_a;
      return $clog2( frac_diff + 1 );

   endfunction

endclass

module scaled_comp_1_behav
  #( int w_c = 5, w_s = 2, w_exp = 5, w_sig = 5, wfp = 1 + w_exp + w_sig )
   ( output logic gt,
     input uwire [wfp-1:0] a, b,
     input uwire [w_s-1:0] s,
     input uwire [w_c-1:0] c );

   logic [wfp-1:0] ssum;

   always_comb begin
      real ra, rb, ssumr;
      ra = conv#(w_exp,w_sig)::ftor( a );
      rb = conv#(w_exp,w_sig)::ftor( b );
      ssumr = ra * 2.0 ** s + rb;
      ssum = conv#(w_exp,w_sig)::rtof( ssumr );
      gt = ssumr > c;
   end

endmodule

// cadence translate_on


////////////////////////////////////////////////////////////////////////////////
/// Testbench Code


// cadence translate_off

function automatic int unsigned rand_wid(int max_wid);
      automatic int wid = 1 + {$random()} % max_wid;
      return {$random()} & ( ( 1 << wid ) - 1 );
endfunction

function automatic real rand_fp(real min, real max);
      automatic real range = max - min;
      localparam real rmax_inv = real'(1) / ( ( longint'(1) << 32 ) - 1 );
      automatic real runit = {$random()} * rmax_inv;
      return runit * range + min;
endfunction

function automatic real fabs(real val);
      fabs = val < 0 ? -val : val;
endfunction

function int min( int a, b );
      min = a <= b ? a : b;
endfunction
function int max( int a, b );
      max = a >= b ? a : b;
endfunction
```

```
virtual class conv2 #(int wexp=6, wsig=10);
   // Convert between real and fp types using parameter-provided
   // exponent and significand sizes.

   localparam int w = 1 + wexp + wsig;
   localparam int bias_r = ( 1 << 11 - 1 ) - 1;
   localparam int w_sig_r = 52;
   localparam int w_exp_r = 11;
   localparam int bias_h = ( 1 << wexp - 1 ) - 1;

   static function logic [w-1:0] rtof( real r );
      logic [wsig-1:0] sig_f;
      logic [w_sig_r-wsig-2:0] sig_x;
      logic sig_x_msb;
      logic [w_exp_r-1:0] exp_r;
      logic sign_r;
      { sign_r, exp_r, sig_f, sig_x_msb, sig_x } = $realtobits(r);
      // So, what about a rounding mode? Not now!
      rtof = !r ? 0 : { sign_r, wexp'( exp_r + bias_h - bias_r ), sig_f };
   endfunction

   static function real ftor( logic [w-1:0] f );
      ftor = !f ? 0.0
        : $bitstoreal
          ( { f[w-1],
              w_exp_r'( bias_r + f[w-2:wsig] - bias_h ),
              f[wsig-1:0], (w_sig_r-wsig)'(0) } );
   endfunction

   static function int err_bits( logic [w-1:0] a, b );

      logic [wsig-1:0] sig_a, sig_b;
      logic [wsig+2:0] frac_a, frac_b, frac_diff;
      logic [wexp-1:0] exp_a, exp_b;
      logic s_a, s_b;
      int delta_e;

      if ( $isunknown(a) || $isunknown(b) ) return 1 << wexp;
      if ( a == b ) return 0;

      { s_a, exp_a, sig_a } = a;
      { s_b, exp_b, sig_b } = b;

      if ( exp_a == 0 || exp_b == 0 ) begin
         logic [wsig-1:0] sig = ~ ( sig_a | sig_b );
         return 1 + wsig - $clog2( sig + 1 );
      end

      delta_e = $abs( 0 + exp_a - exp_b );
      if ( delta_e > 1 ) return delta_e + wsig;
      frac_a = exp_a > exp_b ? { 2'b1, sig_a, 1'b0 } : { 3'b1, sig_a };
      frac_b = exp_b > exp_a ? { 2'b1, sig_b, 1'b0 } : { 3'b1, sig_b };
      frac_diff =
        s_a != s_b ? frac_a + frac_b :
        frac_a > frac_b ? frac_a - frac_b : frac_b - frac_a;
      return $clog2( frac_diff + 1 );

   endfunction

endclass

// cadence translate_on
```

```
// cadence translate_off

// Module names. (Used by the testbench.)
//
typedef enum { M_int, M_fp } M_Type;

module testbench;

   localparam int n_tests = 100000;

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

   localparam int nmsets = 2;
   localparam M_Type mset[2] = '{ M_int, M_fp };

   string mtype_str[M_Type] = '{ M_int: "comp_int", M_fp: "comp_fp " };
   string mtype_abbr[M_Type] = '{ M_int: "in", M_fp: "fp" };

   int t_errs_mod[M_Type];
   int t_errs_each_gt[M_Type][int][2];
   int t_errs_each_ss[M_Type][int][2];
   int t_errs_size_gt[int][2];
   int t_errs_size_ss[int][2];
   int t_errs_each[M_Type][int];

   localparam int nsets = npsets * nmsets;

   logic d[nsets:-1]; // Start / Done signals.

   int t_errs_gt[2], t_errs_ss[2];     // Total number of errors.
   initial begin
      t_errs_gt = '{0,0};
      t_errs_ss = '{0,0};
      for ( int m=0; m<nmsets; m++ )
        for ( int i=0; i<npsets; i++ ) begin
           t_errs_each_gt[mset[m]][i] = '{-1,-1};
           t_errs_each_ss[mset[m]][i] = '{-1,-1};
        end

      d[-1] = 1;
   end

   final begin
   `ifdef xxx
      $write("\nNumber of tests: %0d.\n", n_tests);
      for ( int i=0; i<npsets; i++ )
        $write("Total for exp=%0d, sig=%0d, wc=%2d: Err (s=0): %0d ss, %0d gt.  Err (s>0): %0d ss, %0d gt.\n",
               pset[i][0], pset[i][1], pset[i][2],
               t_errs_size_ss[i][1], t_errs_size_gt[i][1],
               t_errs_size_ss[i][0], t_errs_size_gt[i][0]);
      for ( int i=0; i<nmsets; i++ )
        $write("Total for mod %4s: %5d errors.\n",
               mtype_str[mset[i]],
               t_errs_mod[mset[i]]
               );
      `endif
      for ( int mi=0; mi<nmsets; mi++ )
        for ( int i=0; i<npsets; i++ ) begin
```

```
            automatic M_Type m = mset[mi];
            for ( int j=1; j>=0; j-- )
              $write("Total %s exp=%0d, sig=%0d, wc=%2d, %s: Errors: %0d ss, %0d gt.\n",
                     mtype_str[m],
                     pset[i][0], pset[i][1], pset[i][2], j == 0 ? "s>0" : "s=0",
                     t_errs_each_ss[m][i][j], t_errs_each_gt[m][i][j] );
          end

        //  $write("Total number of errors (s=0): ss %0d, gt %0d.  (s>0): ss %0d, gt %0d.\n",
              //  t_errs_ss[1], t_errs_gt[1], t_errs_ss[0], t_errs_gt[0]);
    end

    for ( genvar m=0; m<nmsets; m++ )
      for ( genvar i=0; i<npsets; i++ ) begin
         localparam int idx = m * npsets + i;
         testbench_n
           #( .w_exp(pset[i][0]), .w_sig(pset[i][1]), .w_c(pset[i][2]),
              .pset(i), .mtype(mset[m]) )
         t2( .done(d[idx]), .tstart(d[idx-1]) );
      end

endmodule


module testbench_n
  #( int w_exp = 5, w_sig = 8, w_c = 12,
     pset = 0, M_Type mtype = M_fp )
   ( output logic done, input uwire tstart );

    // Number of sample outputs to print (whether correct or not).
    localparam int n_samples = 1;

    localparam int w_s = w_exp-1;
    localparam int w_fp = 1 + w_sig + w_exp;
    localparam int bias = ( 1 << w_exp-1 ) - 1;
    localparam int w_sig2 = w_c + 1;
    localparam int w_fp2 = 1 + w_sig2 + w_exp;
    logic [w_c-1:0] c;
    logic [w_s-1:0] sl;
    logic [w_fp-1:0] a, b;
    uwire [w_fp2-1:0] ssum;
    uwire gt;

    localparam int c_max = ( 1 << w_c ) - 1;

    case ( mtype )
      M_fp:
        comp_fp #( w_c, w_s, w_exp, w_sig, w_sig2 )
      c1(gt, ssum, a, b, sl, c);
      M_int:
        comp_int #( w_c, w_s, w_exp, w_sig, w_sig2 )
      c1(gt, ssum, a, b, sl, c);
    endcase

    initial begin

       automatic int n_tests = testbench.n_tests;
       automatic int n_err_gt[2] = '{0,0}, n_err_ss[2] = '{0,0};
       automatic int n_gt = 0;

       wait( tstart );

       $write("\nStarting %4s tests iwth exp=%0d, sig=%0d, wc=%0d\n",
              testbench.mtype_str[mtype], w_exp, w_sig, w_c);
```

```
    for (int i=0; i<n_tests; i++ ) begin

       automatic bit choose_close = $random() & 1'b1;
       automatic bit sl_zero = i < n_tests / 2;
       automatic bit ab_positive = 0;
       logic [w_fp2-1:0] shadow_ssumf;
       int eb_ssum, c_pre;
       real ar, br, shadow_ar, shadow_br, shadow_ssumr, delta, a_scr, ssumr;
       real tol;
       logic shadow_gt;
       bit err_ss, err_gt, err;

       c =  $random() & 7 ? rand_wid(w_c) : c_max;
       sl = sl_zero ? 0 : 1 + rand_wid(w_exp-2);

       ar = ( ab_positive ? rand_fp(0,1):rand_fp(-0.5,0.5) )*rand_wid(w_c+2);

       a_scr = ar * 2 ** sl;

       br = choose_close
         ? c - a_scr + rand_fp(-1,1)
         : ( ab_positive ? rand_fp(0,1):rand_fp(-0.5,0.5) ) * rand_wid(w_c+2);

       if ( ab_positive && br < 0 ) br = -br;

       a = conv#(w_exp,w_sig)::rtof( ar );
       b = conv#(w_exp,w_sig)::rtof( br );

       shadow_ar = conv#(w_exp,w_sig)::ftor( a );
       shadow_br = conv#(w_exp,w_sig)::ftor( b );
       shadow_ssumr = shadow_ar * 2.0**sl + shadow_br;
       shadow_ssumf = conv#(w_exp,w_sig2)::rtof(shadow_ssumr);
       delta = shadow_ssumr - c;
       tol = c / real'( 1 << w_sig2 + 1 );
       shadow_gt = shadow_ssumr > c;
       n_gt += shadow_gt;

       #1;

       ssumr = conv#(w_exp,w_sig2)::ftor(ssum);
       err_gt = fabs(delta) > tol && gt !== shadow_gt;
       eb_ssum = conv#(w_exp,w_sig2)::err_bits(ssum,shadow_ssumf);
       err_ss = eb_ssum > 1;
       err = err_gt || err_ss;

       if ( i < n_samples || err ) begin
          if ( err_gt ) n_err_gt[sl_zero]++;
          if ( err_ss ) n_err_ss[sl_zero]++;

          if ( i < n_samples || err_ss && n_err_ss[sl_zero] < 5 )
            $write( "%s %s #(%0d,%0d,%0d) a=%.2f, s=%0d, b=%.2f, c=%0d. ss %.4e %s %.4e (correct)\n",
                    err_ss ? "Error " : "Sample",
                    testbench.mtype_abbr[mtype],
                    w_exp, w_sig, w_c,
                    shadow_ar, sl, shadow_br, c,
                    ssumr, err_ss ? "!=" : "==", shadow_ssumr );

          if ( i < n_samples || err_gt && n_err_gt[sl_zero] < 5 )
            $write( "%s %s #(%0d,%0d,%0d) a=%.2f, s=%0d, b=%.2f, c=%0d. gt %h %s %h (correct) %g\n",
                    err_gt ? "Error " : "Sample",
                    testbench.mtype_abbr[mtype],
                    w_exp, w_sig, w_c,
                    shadow_ar, sl, shadow_br, c,
                    gt, err_gt ? "!=" : "==", shadow_gt, delta );
       end
```

```
      end

      for ( int i=1; i>=0; i-- )
        $write("Finished  %4s tests exp=%0d, sig=%0d, wc=%0d, %s. Errors: %0d ss, %0d gt\n",
             testbench.mtype_str[mtype], w_exp, w_sig, w_c,
               i == 0 ? "s>0" : "s=0",
               n_err_ss[i], n_err_gt[i]);
      //  $write("Frac gt %.4f\n", real'(n_gt)/n_tests);

      for ( int i=0; i<2; i++ ) begin
         testbench.t_errs_gt[i] += n_err_gt[i];
         testbench.t_errs_ss[i] += n_err_ss[i];
         testbench.t_errs_each_gt[mtype][pset][i] = n_err_gt[i];
         testbench.t_errs_each_ss[mtype][pset][i] = n_err_ss[i];
      end

      done = 1;
   end

endmodule

// Define SIMULATION_ON in a translate_off region. Used to control
// whether simulation or synthesis versions of Chipware modules are
// included.
//
`define SIMULATION_ON
//
// cadence translate_on


`default_nettype wire

`ifdef SIMULATION_ON

`include "/apps/linux/cadence/DDIEXPORT23/GENUS231/share/synth/lib/chipware/sim/verilog/CW/CW_fp_mult.v"
`include "/apps/linux/cadence/DDIEXPORT23/GENUS231/share/synth/lib/chipware/sim/verilog/CW/CW_fp_add.v"
`include "/apps/linux/cadence/DDIEXPORT23/GENUS231/share/synth/lib/chipware/sim/verilog/CW/CW_fp_sub.v"
`include "/apps/linux/cadence/DDIEXPORT23/GENUS231/share/synth/lib/chipware/sim/verilog/CW/CW_fp_div.v"
`include "/apps/linux/cadence/DDIEXPORT23/GENUS231/share/synth/lib/chipware/sim/verilog/CW/CW_fp_i2flt.v"
`include "/apps/linux/cadence/DDIEXPORT23/GENUS231/share/synth/lib/chipware/sim/verilog/CW/CW_fp_flt2i.v"
`include "/apps/linux/cadence/DDIEXPORT23/GENUS231/share/synth/lib/chipware/sim/verilog/CW/CW_fp_cmp.v"

`else

`include "/apps/linux/cadence/DDIEXPORT23/GENUS231/share/synth/lib/chipware/syn/CW/CW_fp_mult.v"
`include "/apps/linux/cadence/DDIEXPORT23/GENUS231/share/synth/lib/chipware/syn/CW/CW_fp_add.v"
`include "/apps/linux/cadence/DDIEXPORT23/GENUS231/share/synth/lib/chipware/syn/CW/CW_fp_sub.v"
`include "/apps/linux/cadence/DDIEXPORT23/GENUS231/share/synth/lib/chipware/syn/CW/CW_fp_i2flt.v"
`include "/apps/linux/cadence/DDIEXPORT23/GENUS231/share/synth/lib/chipware/syn/CW/CW_fp_flt2i.v"
`include "/apps/linux/cadence/DDIEXPORT23/GENUS231/share/synth/lib/chipware/syn/CW/CW_fp_div.v"
`include "/apps/linux/cadence/DDIEXPORT23/GENUS231/share/synth/lib/chipware/syn/CW/CW_fp_cmp.v"

`endif
```

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2024 Homework 3 -- SOLUTION Prob 1
//

 /// Assignment https://www.ece.lsu.edu/koppel/v/2024/hw03.pdf

`default_nettype none

 /// Convenient Character Names
//
typedef enum logic [3:0]
  { Char_Blank, Char_Dot,
    Char_Open, Char_Close,
    Char_Open_Okay, Char_Close_Okay } Char;


////////////////////////////////////////////////////////////////////////////////
/// Problem 1
//
  /// Complete pmatch_base.
//
//     [✔] The module can use procedural code.
//     [✔] The module must a recursive and describe tree-structured hardware.
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//
//     [✔] Don't assume any particular parameter values.
//
//     [✔] As always, code must be written clearly.
//     [✔] As always, pay attention to cost and performance.


module pmatch_base
  #( int n = 5, wn = $clog2(n+1) )
   ( output logic [wn-1:0] left_out_n_unmat_close, right_out_n_unmat_open,
     input uwire [3:0] str[0:n-1] );

   /// SOLUTION

   if ( n == 1 ) begin

      // There is only one character, so if it's a parenthesis it must
      // be unmatched.
      //
      assign left_out_n_unmat_close = str[0] == Char_Close ? 1 : 0;
      assign right_out_n_unmat_open = str[0] == Char_Open ? 1 : 0;

   end else begin

      // Split the string between recursive instantiations.
      //
```

```
      localparam int n_left = n/2;
      localparam int n_right = n - n_left;
      localparam int wl = $clog2(n_left+1),  wr = $clog2(n_right+1);
      //
      uwire [wl-1:0] lt_close, lt_open;
      uwire [wr-1:0] rt_close, rt_open;
      //
      pmatch_base #(n_left,  wl) plt( lt_close, lt_open, str[0:n_left-1] );
      pmatch_base #(n_right, wr) prt( rt_close, rt_open, str[n_left:n-1] );

      // Compute the number of remaining unmatched opening parentheses
      // when the lt_open unmatched parentheses from plt are matched
      // with the rt_close closing parentheses from prt.
      //
      uwire logic signed [wn-1:0] delta = lt_open - rt_close;
      //
      // If delta is positive there are leftover unmatched opening
      // parentheses that need to be added to output
      // right_out_n_unmat_open, and if delta is negative there are
      // leftover unmatched closing parentheses that need to be added
      // to output left_out_n_unmat_close.
      //
      // Note: Declaring delta as a *signed* type is necessary so that
      // the comparisons below, such as "delta >= 0" are correct. If
      // delta were unsigned then "delta >= 0" would always be true!
      //
      assign right_out_n_unmat_open = delta >= 0 ? rt_open  + delta : rt_open;
      assign left_out_n_unmat_close = delta <  0 ? lt_close - delta : lt_close;

   end

endmodule



module pmatch_comb_base
  #( int n = 5, wn = $clog2(n) )
   ( output logic [wn-1:0] left_out_n_unmat_close, right_out_n_unmat_open,
     input uwire [3:0] str[0:n-1] );

   /// Reference Module
   //
   //  Examine this module to help understand how to solve the problem.
   //
   //  The solution should NOT BE placed in this module.

   always_comb begin

      left_out_n_unmat_close = 0;
      right_out_n_unmat_open = 0;

      // Scan string from left to right. (Leftmost character is at i=0.)
      for ( int i=0; i<n; i++ )
```

```
        if ( str[i] == Char_Close ) begin

           // We've found a closing parenthesis, ")".

           if ( right_out_n_unmat_open > 0 )
             right_out_n_unmat_open--;  // It matches an opening parenthesis.
           else
             left_out_n_unmat_close++;  // It's unmatched, update the count.

        end else if ( str[i] == Char_Open ) begin

           // We've found an opening parenthesis, "(".

           // Increment the count of unmatched open parentheses ..
           right_out_n_unmat_open++;
           // .. which might be decremented in a later iteration.

        end
   end

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Problem 2
//
  /// Complete pmatch_mark
//
//     [ ] The module can use procedural code.
//     [ ] The module must a recursive and describe tree-structured hardware.
//
//     [ ] Make sure that the testbench does not report errors.
//     [ ] Module must be synthesizable. Use command: genus -files syn.tcl
//
//     [ ] Don't assume any particular parameter values.
//
//     [ ] As always, code must be written clearly.
//     [ ] As always, pay attention to cost and performance.



module pmatch_mark
  #( int n = 5, wn = $clog2(n+1) )
   ( output logic [wn-1:0] left_out_n_unmat_close, right_out_n_unmat_open,
     output uwire [3:0] str_marked [0:n-1],
     input uwire [wn-1:0] left_in_n_unmat_open, right_in_n_unmat_close,
     input uwire [3:0] str [0:n-1] );


endmodule



module pmatch_comb_mark
```

```
#( int n = 5, wn = $clog2(n+1) )
 ( output logic [wn-1:0] left_out_n_unmat_close, right_out_n_unmat_open,
   output logic [3:0] str_marked [0:n-1],
   input uwire [wn-1:0] left_in_n_unmat_open, right_in_n_unmat_close,
   input uwire [3:0] str [0:n-1] );

 /// Reference Module
 //
 //  Examine this module to help understand how to solve the problem.
 //
 //  The solution should NOT BE placed in this module.

 always_comb begin
    automatic logic [wn-1:0] n_um_op = left_in_n_unmat_open;
    automatic logic [wn-1:0] n_um_cl = right_in_n_unmat_close;
    left_out_n_unmat_close = 0;
    right_out_n_unmat_open = 0;
    str_marked = str;

    // Scan string from left to right. (Leftmost character is at i=0.)
    //
    // The loop below is identical to the one in pmatch_comb_base.
    //
    for ( int i=0; i<n; i++ )

      if ( str[i] == Char_Close ) begin

         // We've found a closing parenthesis, ")".

         if ( right_out_n_unmat_open > 0 )
           right_out_n_unmat_open--;  // It matches an opening parenthesis.
         else
           left_out_n_unmat_close++;  // It's unmatched, update the count.

      end else if ( str[i] == Char_Open ) begin

         // We've found an opening parenthesis, "(".

         // Increment the count of unmatched open parentheses ..
         right_out_n_unmat_open++;
         // .. which might be decremented in a later iteration.

      end

    // Scan string forward and mark matched closing parentheses.
    //
    for ( int i=0; i<n; i++ )
      if ( str[i] == Char_Close  &&  n_um_op > 0 ) begin

         n_um_op--;
         str_marked[i] = Char_Close_Okay;

      end else if ( str[i] == Char_Open ) begin
```

```
            n_um_op++;

          end

        // Scan string backward and mark matched opening parentheses.
        //
        for ( int i=n-1; i>=0; i-- )
          if ( str[i] == Char_Open  &&  n_um_cl > 0 ) begin

            n_um_cl--;
            str_marked[i] = Char_Open_Okay;

          end else if ( str[i] == Char_Close ) begin

            n_um_cl++;

          end
    end

endmodule
```

```
////////////////////////////////////////////////////////////////////////////////
```

/// Testbench Code

```
// cadence translate_off

// Module names. (Used by the testbench.)
//
typedef enum { M_base, M_mark } M_Type;

module testbench;

    localparam int n_tests = 100000;

    localparam int npsets = 6; // This MUST be set to the size of pset.
    localparam int pset[npsets][1] =
               '{{ 4 }, { 5 }, { 7 }, { 8 }, { 9 }, { 17 }};

    localparam int nmsets = 2;
    localparam M_Type mset[2] = '{ M_base, M_mark };

    string mtype_str[M_Type] =
           '{ M_base: "pmatch_base", M_mark: "pmatch_mark" };
    string mtype_abbr[M_Type] = '{ M_base: "base", M_mark: "mark" };

    int t_errs_each_cl[M_Type][int];
    int t_errs_each_op[M_Type][int];
    int t_errs_each_mk[M_Type][int];
    int t_errs_cl, t_errs_op, t_errs_mk;
```

```
   localparam int nsets = npsets * nmsets;

   logic d[nsets:-1]; // Start / Done signals.

   initial begin
      t_errs_cl = 0;
      t_errs_op = 0; t_errs_mk = 0;
      for ( int m=0; m<nmsets; m++ )
        for ( int i=0; i<npsets; i++ ) begin
           automatic int n = pset[i][0];
           t_errs_each_cl[mset[m]][n] = 0;
           t_errs_each_op[mset[m]][n] = 0;
           t_errs_each_mk[mset[m]][n] = 0;
        end

      d[-1] = 1;
   end

   final begin
      for ( int mi=0; mi<nmsets; mi++ )
        for ( int i=0; i<npsets; i++ ) begin
           automatic M_Type m = mset[mi];
           automatic int n = pset[i][0];
           $write("Total %s n=%0d: Errors: %0d cl, %0d op, %0d mk.\n",
                  mtype_str[m], n,
                  t_errs_each_cl[m][n],
                  t_errs_each_op[m][n],
                  t_errs_each_mk[m][n]);
        end

   end

   for ( genvar m=0; m<nmsets; m++ )
     for ( genvar i=0; i<npsets; i++ ) begin
        localparam int idx = m * npsets + i;
        testbench_n
          #( .n(pset[i][0]), .mtype(mset[m]) )
        t2( .done(d[idx]), .tstart(d[idx-1]) );
     end

endmodule



module testbench_n
  #( int n = 4, M_Type mtype = M_base )
   ( output logic done, input uwire tstart );

   localparam int wn = $clog2(n+1);
   localparam int n_tests = 2000;

   localparam int n_samples_show = 10;
   localparam int n_errors_show = 5;
```

```
uwire [wn-1:0] n_cl_lt, n_op_rt;
logic [wn-1:0] n_u_op, n_u_cl;
logic [3:0] str[0:n-1];
uwire [3:0] str_marked[0:n-1];

string char_to_ascii[logic[3:0]] =
       '{ Char_Blank: " ",
          Char_Dot: ".",
          Char_Open: "(",
          Char_Open_Okay: "<",
          Char_Close: ")",
          Char_Close_Okay: ">" };
logic [3:0] ascii_to_char[string];

function automatic string char_to_string( logic [3:0] str[0:n-1] );
   automatic string str_txt = "";
   for ( int j=0; j<n; j++ ) str_txt = { str_txt, char_to_ascii[str[j]] };
   char_to_string = str_txt;
endfunction

case ( mtype )
  M_base:
    pmatch_base #(n,wn) pm( n_cl_lt, n_op_rt, str );
    // pmatch_comb_base #(n,wn) pcomb( n_cl_lt, n_op_rt, str );
  M_mark:
    pmatch_mark #(n,wn) pm(n_cl_lt,n_op_rt,str_marked,n_u_op,n_u_cl, str );
    //  pmatch_comb_mark #(n,wn) pm( n_cl_lt, n_op_rt, str_marked, n_u_op, n_u_cl, str );
endcase



string xstr_special[] =
       '{ "))", "((", "()", ")(", ")()(", ")()", "", ")", "(",
          ") )", "( (", " ) )", " ( (", ") ) ", "( ( ", "( )", ") (" };
string str_special[] =
       '{ "()", ".( )", ")(",  ")", "))", "()))", "())(", "())(((",
         "))(((", ")", "(", "))", "((", "()", ")()(", ")()" };

initial begin

   automatic int n_errs_cl =0, n_errs_op = 0, n_errs_mk = 0;
   automatic int n_samples = 0;
   automatic string prefix_txt =
     $sformatf("%s n=%0d",testbench.mtype_str[mtype],n);
   automatic string prefix_txt_str;

   foreach ( char_to_ascii[c] ) ascii_to_char[char_to_ascii[c]] = c;

   wait( tstart );

   $write("Starting %s tests for n=%0d.\n",
          testbench.mtype_str[mtype], n);
```

```
    for ( int i=0; i<n_tests; i++ ) begin
       automatic int shadow_n_close_lt, shadow_n_open_rt;
       automatic int n_unm_op, n_unm_cl;
       automatic string str_txt;
       automatic logic [3:0] shadow_str_e[0:n-1];
       automatic bit err_cl, err_op, err_mk, err;

       n_u_op = 0;
       n_u_cl = 0;

       if ( i < str_special.size() ) begin

          automatic string spc = str_special[i];
          foreach ( spc[j] ) str[j] = ascii_to_char[spc[j]];
          for ( int j=spc.len(); j<n; j++ ) str[j] = Char_Blank;

       end else begin

          for ( int j=0; j<n; j++ ) str[j] = (4)'({$random} % 4);

       end

       str_txt = char_to_string( str );

       shadow_n_close_lt = 0;
       shadow_n_open_rt = 0;
       for ( int j=0; j<n; j++ ) begin
          if ( str[j] == Char_Close ) begin
            if ( shadow_n_open_rt ) shadow_n_open_rt--;
            else shadow_n_close_lt++;
          end
          if ( str[j] == Char_Open ) shadow_n_open_rt++;
       end

       n_unm_op = n_u_op;
       shadow_str_e = str;
       for ( int j=0; j<n; j++ ) begin
          if ( str[j] == Char_Close ) begin
             if ( n_unm_op > 0 ) begin
                shadow_str_e[j] = Char_Close_Okay;
                n_unm_op--;
             end
          end
          if ( str[j] == Char_Open ) n_unm_op++;
       end
       n_unm_cl = n_u_cl;
       for ( int j=n-1; j>=0; j-- ) begin
          if ( str[j] == Char_Open ) begin
             if ( n_unm_cl > 0 ) begin
                shadow_str_e[j] = Char_Open_Okay;
                n_unm_cl--;
             end
          end
```

```
      if ( str[j] == Char_Close ) n_unm_cl++;
   end

   #1;

   err_cl = n_cl_lt !== shadow_n_close_lt;
   err_op = n_op_rt !== shadow_n_open_rt;
   err_mk = mtype == M_mark && str_marked !== shadow_str_e;
   err = err_cl || err_op || err_mk;

   prefix_txt_str = $sformatf("%s '%s'",prefix_txt,str_txt);

   if ( !err && n_samples < n_samples_show ) begin
      n_samples++;
      $write("Sample %s: close = %0d,  open = %0d (both correct)\n",
             prefix_txt_str, n_cl_lt, n_op_rt);

      if ( mtype == M_mark )
        $write("Sample %s '%s' (marked_outpuut)\n",
               prefix_txt, char_to_string(shadow_str_e));
   end

   if ( err_cl ) begin
      n_errs_cl++;
      if ( n_errs_cl < n_errors_show )
        $write("Error %s: close  %0d != %0d (correct)\n",
               prefix_txt_str, n_cl_lt, shadow_n_close_lt);
   end
   if ( err_op ) begin
      n_errs_op++;
      if ( n_errs_op < n_errors_show )
        $write("Error %s: open   %0d != %0d (correct)\n",
               prefix_txt_str, n_op_rt, shadow_n_open_rt);
   end

   if ( err_mk ) begin
      n_errs_mk++;
      if ( n_errs_mk < n_errors_show ) begin
         $write("Error %s: '%s' != '%s' (correct)\n",
                prefix_txt,
                char_to_string(str_marked), char_to_string(shadow_str_e));
      end

   end

end

$write("Done with tests %s. Errors: %0d cl, %0d op, %0d mark.\n",
       prefix_txt,
       n_errs_cl, n_errs_op, n_errs_mk);

testbench.t_errs_each_cl[mtype][n] = n_errs_cl;
testbench.t_errs_each_op[mtype][n] = n_errs_op;
testbench.t_errs_each_mk[mtype][n] = n_errs_mk;
```

```
      done = 1;
    end
endmodule
// cadence translate_on
```

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2024 Homework 3 -- SOLUTION
//

 /// Assignment https://www.ece.lsu.edu/koppel/v/2024/hw03.pdf

`default_nettype none

 /// Convenient Character Names
//
typedef enum logic [3:0]
  { Char_Blank, Char_Dot,
    Char_Open, Char_Close,
    Char_Open_Okay, Char_Close_Okay } Char;


////////////////////////////////////////////////////////////////////////////////
/// Problem 1
//
  /// Complete pmatch_base.
//
//     [✔] The module can use procedural code.
//     [✔] The module must a recursive and describe tree-structured hardware.
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//
//     [✔] Don't assume any particular parameter values.
//
//     [✔] As always, code must be written clearly.
//     [✔] As always, pay attention to cost and performance.


module pmatch_base
  #( int n = 5, wn = $clog2(n+1) )
   ( output logic [wn-1:0] left_out_n_unmat_close, right_out_n_unmat_open,
     input uwire [3:0] str[0:n-1] );

   /// SOLUTION

   if ( n == 1 ) begin

      assign left_out_n_unmat_close = str[0] == Char_Close;
      assign right_out_n_unmat_open = str[0] == Char_Open;

   end else begin

      localparam int n_left = n/2;
      localparam int n_right = n - n_left;
      localparam int wl = $clog2(n_left+1),  wr = $clog2(n_right+1);

      uwire [wl-1:0] lt_close, lt_open;
```

```
      uwire [wr-1:0] rt_close, rt_open;

      pmatch_base #(n_left,  wl) plt( lt_close, lt_open, str[0:n_left-1] );
      pmatch_base #(n_right, wr) prt( rt_close, rt_open, str[n_left:n-1] );

      uwire logic signed [wn-1:0] delta = lt_open - rt_close;
      assign left_out_n_unmat_close = delta >= 0 ? lt_close : lt_close - delta;
      assign right_out_n_unmat_open = delta < 0  ? rt_open  : rt_open + delta;

   end

endmodule



module pmatch_comb_base
  #( int n = 5, wn = $clog2(n) )
   ( output logic [wn-1:0] left_out_n_unmat_close, right_out_n_unmat_open,
     input uwire [3:0] str[0:n-1] );

   /// Reference Module
   //
   //  Examine this module to help understand how to solve the problem.
   //
   //  The solution should NOT BE placed in this module.

   always_comb begin

      left_out_n_unmat_close = 0;
      right_out_n_unmat_open = 0;

      // Scan string from left to right. (Leftmost character is at i=0.)
      for ( int i=0; i<n; i++ )

        if ( str[i] == Char_Close ) begin

           // We've found a closing parenthesis, ")".

           if ( right_out_n_unmat_open > 0 )
             right_out_n_unmat_open--;  // It matches an opening parenthesis.
           else
             left_out_n_unmat_close++;  // It's unmatched, update the count.

        end else if ( str[i] == Char_Open ) begin

           // We've found an opening parenthesis, "(".

           // Increment the count of unmatched open parentheses ..
           right_out_n_unmat_open++;
           // .. which might be decremented in a later iteration.

        end
   end
```

```
endmodule


////////////////////////////////////////////////////////////////////////////////
/// Problem 2
//
  /// Complete pmatch_mark
//
//     [✔] The module can use procedural code.
//     [✔] The module must a recursive and describe tree-structured hardware.
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//
//     [✔] Don't assume any particular parameter values.
//
//     [✔] As always, code must be written clearly.
//     [✔] As always, pay attention to cost and performance.


module pmatch_mark
  #( int n = 5, wn = $clog2(n+1) )
   ( output logic [wn-1:0] left_out_n_unmat_close, right_out_n_unmat_open,
     output uwire [3:0] str_marked [0:n-1],
     input uwire [wn-1:0] left_in_n_unmat_open, right_in_n_unmat_close,
     input uwire [3:0] str [0:n-1] );

   /// SOLUTION

   if ( n == 1 ) begin

      assign left_out_n_unmat_close = str[0] == Char_Close;
      assign right_out_n_unmat_open = str[0] == Char_Open;

      assign str_marked[0] =
             str[0] == Char_Close && left_in_n_unmat_open   ? Char_Close_Okay :
             str[0] == Char_Open  && right_in_n_unmat_close ? Char_Open_Okay  :
             str[0];

   end else begin

      localparam int n_left = n/2;
      localparam int n_right = n - n_left;
      localparam int wl = $clog2( n_left+1 );
      localparam int wr = $clog2( n_right+1 );

      localparam logic [wl-1:0] nl_max = ~(wl)'(0); // All 1's.
      localparam logic [wr-1:0] nr_max = ~(wr)'(0); // All 1's.

      `define min(a,limit) ( unsigned'(a) <= (limit) ? a : limit )

      // Recursive Instance Outputs
      //
      uwire [wl-1:0] lt_close, lt_open;
```

```
uwire [wr-1:0] rt_close, rt_open;

// Same as Problem 1
//
uwire signed [wn-1:0] delta = lt_open - rt_close;
assign left_out_n_unmat_close = delta >= 0 ? lt_close : lt_close - delta;
assign right_out_n_unmat_open = delta < 0  ? rt_open  : rt_open + delta;

/// Problem 2
//
// Pass the incoming number of matched parentheses to the
// recursive instances. Use min in case their values are too
// large.
//
uwire [wl-1:0] lt_matched_cl = `min( left_in_n_unmat_open,   nl_max );
uwire [wr-1:0] rt_matched_op = `min( right_in_n_unmat_close, nr_max );

/// Compute Right Recursive Instance Input rt_matched_cl
//
// Compute the number of closing parentheses that could be
// matched in the right recursive instance (rt_matched_cl) using
// the incoming unmatched opening parentheses and unmatched
// opening parentheses found in the left instance.
//
// First, compute more_op: the number of unmatched incoming open
// parentheses that were not matched by the portion of str
// connected to the left recursive instance (plt).
//
uwire signed [wn:0] more_op = left_in_n_unmat_open - lt_close;
//
// Then, compute the number of unmatched opening parentheses to
// the left of position str[n_left], accounting for the opening
// parentheses found by the left recursive instance (lt_open)
// and still unmatched incoming open parentheses (more_op).
//
uwire [wr-1:0] rt_matched_cl =
       more_op < 0  ? lt_open  : `min( lt_open + more_op, nr_max );

/// Compute Left Recursive Instance Input lt_matched_op
//
// Similar to the procedure above, compute the number of still
// unmatched incoming closing parentheses (more_cl) and combine
// it with the number of closing parentheses found in the right
// recursive instance.
//
uwire signed [wn:0] more_cl = right_in_n_unmat_close - rt_open;
uwire [wl-1:0] lt_matched_op =
       more_cl < 0  ? rt_close : `min( rt_close + more_cl, nl_max );

/// Recursive Instantiations
//
pmatch_mark #(n_left,wl)
plt( lt_close, lt_open, str_marked[0:n_left-1],
     lt_matched_cl, lt_matched_op, str[0:n_left-1] );
```

```
      pmatch_mark #(n_right,wr)
      prt( rt_close, rt_open, str_marked[n_left:n-1],
           rt_matched_cl, rt_matched_op, str[n_left:n-1] );

   end

endmodule



module pmatch_comb_mark
  #( int n = 5, wn = $clog2(n+1) )
   ( output logic [wn-1:0] left_out_n_unmat_close, right_out_n_unmat_open,
     output logic [3:0] str_marked [0:n-1],
     input uwire [wn-1:0] left_in_n_unmat_open, right_in_n_unmat_close,
     input uwire [3:0] str [0:n-1] );

   /// Reference Module
   //
   //  Examine this module to help understand how to solve the problem.
   //
   //  The solution should NOT BE placed in this module.

   always_comb begin
      automatic logic [wn-1:0] n_um_op = left_in_n_unmat_open;
      automatic logic [wn-1:0] n_um_cl = right_in_n_unmat_close;
      left_out_n_unmat_close = 0;
      right_out_n_unmat_open = 0;
      str_marked = str;

      // Scan string from left to right. (Leftmost character is at i=0.)
      //
      // The loop below is identical to the one in pmatch_comb_base.
      //
      for ( int i=0; i<n; i++ )

        if ( str[i] == Char_Close ) begin

           // We've found a closing parenthesis, ")".

           if ( right_out_n_unmat_open > 0 )
             right_out_n_unmat_open--;  // It matches an opening parenthesis.
           else
             left_out_n_unmat_close++;  // It's unmatched, update the count.

        end else if ( str[i] == Char_Open ) begin

           // We've found an opening parenthesis, "(".

           // Increment the count of unmatched open parentheses ..
           right_out_n_unmat_open++;
           // .. which might be decremented in a later iteration.

        end
```

```
      // Scan string forward and mark matched closing parentheses.
      //
      for ( int i=0; i<n; i++ )
        if ( str[i] == Char_Close  &&  n_um_op > 0 ) begin

           n_um_op--;
           str_marked[i] = Char_Close_Okay;

        end else if ( str[i] == Char_Open ) begin

           n_um_op++;

        end

      // Scan string backward and mark matched opening parentheses.
      //
      for ( int i=n-1; i>=0; i-- )
        if ( str[i] == Char_Open  &&  n_um_cl > 0 ) begin

           n_um_cl--;
           str_marked[i] = Char_Open_Okay;

        end else if ( str[i] == Char_Close ) begin

           n_um_cl++;

        end
   end

endmodule




////////////////////////////////////////////////////////////////////////////////
/// Testbench Code


// cadence translate_off

// Module names. (Used by the testbench.)
//
typedef enum { M_base, M_mark } M_Type;

module testbench;

   localparam int n_tests = 100000;

   localparam int npsets = 6; // This MUST be set to the size of pset.
   localparam int pset[npsets][1] =
              '{{ 4 }, { 5 }, { 7 }, { 8 }, { 9 }, { 17 }};

   localparam int nmsets = 2;
```

```
   localparam M_Type mset[2] = '{ M_base, M_mark };

   string mtype_str[M_Type] =
          '{ M_base: "pmatch_base", M_mark: "pmatch_mark" };
   string mtype_abbr[M_Type] = '{ M_base: "base", M_mark: "mark" };

   int t_errs_each_cl[M_Type][int];
   int t_errs_each_op[M_Type][int];
   int t_errs_each_mk[M_Type][int];
   int t_errs_cl, t_errs_op, t_errs_mk;

   localparam int nsets = npsets * nmsets;

   logic d[nsets:-1]; // Start / Done signals.

   initial begin
      t_errs_cl = 0;
      t_errs_op = 0; t_errs_mk = 0;
      for ( int m=0; m<nmsets; m++ )
        for ( int i=0; i<npsets; i++ ) begin
           automatic int n = pset[i][0];
           t_errs_each_cl[mset[m]][n] = 0;
           t_errs_each_op[mset[m]][n] = 0;
           t_errs_each_mk[mset[m]][n] = 0;
        end

      d[-1] = 1;
   end

   final begin
      for ( int mi=0; mi<nmsets; mi++ )
        for ( int i=0; i<npsets; i++ ) begin
           automatic M_Type m = mset[mi];
           automatic int n = pset[i][0];
           $write("Total %s n=%0d: Errors: %0d cl, %0d op, %0d mk.\n",
                  mtype_str[m], n,
                  t_errs_each_cl[m][n],
                  t_errs_each_op[m][n],
                  t_errs_each_mk[m][n]);
        end

   end

   for ( genvar m=0; m<nmsets; m++ )
     for ( genvar i=0; i<npsets; i++ ) begin
        localparam int idx = m * npsets + i;
        testbench_n
          #( .n(pset[i][0]), .mtype(mset[m]) )
        t2( .done(d[idx]), .tstart(d[idx-1]) );
     end

endmodule
```

```
module testbench_n
  #( int n = 4, M_Type mtype = M_base )
   ( output logic done, input uwire tstart );

   localparam int wn = $clog2(n+1);
   localparam int n_tests = 2000;

   localparam int n_samples_show = 10;
   localparam int n_errors_show = 5;



   uwire [wn-1:0] n_cl_lt, n_op_rt;
   logic [wn-1:0] n_u_op, n_u_cl;
   logic [3:0] str[0:n-1];
   uwire [3:0] str_marked[0:n-1];

   string char_to_ascii[logic[3:0]] =
          '{ Char_Blank: " ",
             Char_Dot: ".",
             Char_Open: "(",
             Char_Open_Okay: "<",
             Char_Close: ")",
             Char_Close_Okay: ">" };
   logic [3:0] ascii_to_char[string];

   function automatic string char_to_string( logic [3:0] str[0:n-1] );
      automatic string str_txt = "";
      for ( int j=0; j<n; j++ ) str_txt = { str_txt, char_to_ascii[str[j]] };
      char_to_string = str_txt;
   endfunction

   case ( mtype )
     M_base:
       pmatch_base #(n,wn) pm( n_cl_lt, n_op_rt, str );
       // pmatch_comb_base #(n,wn) pcomb( n_cl_lt, n_op_rt, str );
     M_mark:
       pmatch_mark #(n,wn) pm(n_cl_lt,n_op_rt,str_marked,n_u_op,n_u_cl, str );
       //  pmatch_comb_mark #(n,wn) pm( n_cl_lt, n_op_rt, str_marked, n_u_op, n_u_cl, str );
   endcase



   string xstr_special[] =
          '{ "))", "((", "()", ")(", ")()(", ")()", "", ")", "(",
             ") )", "( (", " ) )", " ( (", ") ) ", "( ( ", "( )", ") (" };
   string str_special[] =
          '{ "()", ".( )", ")(",  ")", "))", "()))", "())(", "())(((",
            "))((", ")", "(", "))", "((", "()", ")()(", ")()" };

   initial begin

      automatic int n_errs_cl =0, n_errs_op = 0, n_errs_mk = 0;
```

```
    automatic int n_samples = 0;
    automatic string prefix_txt =
      $sformatf("%s n=%0d",testbench.mtype_str[mtype],n);
    automatic string prefix_txt_str;

    foreach ( char_to_ascii[c] ) ascii_to_char[char_to_ascii[c]] = c;

    wait( tstart );

    $write("Starting %s tests for n=%0d.\n",
           testbench.mtype_str[mtype], n);

    for ( int i=0; i<n_tests; i++ ) begin
       automatic int shadow_n_close_lt, shadow_n_open_rt;
       automatic int n_unm_op, n_unm_cl;
       automatic string str_txt;
       automatic logic [3:0] shadow_str_e[0:n-1];
       automatic bit err_cl, err_op, err_mk, err;

       n_u_op = 0;
       n_u_cl = 0;

       if ( i < str_special.size() ) begin

          automatic string spc = str_special[i];
          foreach ( spc[j] ) str[j] = ascii_to_char[spc[j]];
          for ( int j=spc.len(); j<n; j++ ) str[j] = Char_Blank;

       end else begin

          for ( int j=0; j<n; j++ ) str[j] = (4)'({$random} % 4);

       end

       str_txt = char_to_string( str );

       shadow_n_close_lt = 0;
       shadow_n_open_rt = 0;
       for ( int j=0; j<n; j++ ) begin
          if ( str[j] == Char_Close ) begin
            if ( shadow_n_open_rt ) shadow_n_open_rt--;
            else shadow_n_close_lt++;
          end
          if ( str[j] == Char_Open ) shadow_n_open_rt++;
       end

       n_unm_op = n_u_op;
       shadow_str_e = str;
       for ( int j=0; j<n; j++ ) begin
          if ( str[j] == Char_Close ) begin
             if ( n_unm_op > 0 ) begin
                shadow_str_e[j] = Char_Close_Okay;
                n_unm_op--;
             end
```

```
   end
   if ( str[j] == Char_Open ) n_unm_op++;
end
n_unm_cl = n_u_cl;
for ( int j=n-1; j>=0; j-- ) begin
   if ( str[j] == Char_Open ) begin
      if ( n_unm_cl > 0 ) begin
         shadow_str_e[j] = Char_Open_Okay;
         n_unm_cl--;
      end
   end
   if ( str[j] == Char_Close ) n_unm_cl++;
end

#1;

err_cl = n_cl_lt !== shadow_n_close_lt;
err_op = n_op_rt !== shadow_n_open_rt;
err_mk = mtype == M_mark && str_marked !== shadow_str_e;
err = err_cl || err_op || err_mk;

prefix_txt_str = $sformatf("%s '%s'",prefix_txt,str_txt);

if ( !err && n_samples < n_samples_show ) begin
   n_samples++;
   $write("Sample %s: close = %0d,  open = %0d (both correct)\n",
          prefix_txt_str, n_cl_lt, n_op_rt);

   if ( mtype == M_mark )
     $write("Sample %s '%s' (marked_outpuut)\n",
            prefix_txt, char_to_string(shadow_str_e));
end

if ( err_cl ) begin
   n_errs_cl++;
   if ( n_errs_cl < n_errors_show )
     $write("Error %s: close  %0d != %0d (correct)\n",
            prefix_txt_str, n_cl_lt, shadow_n_close_lt);
end
if ( err_op ) begin
   n_errs_op++;
   if ( n_errs_op < n_errors_show )
     $write("Error %s: open   %0d != %0d (correct)\n",
            prefix_txt_str, n_op_rt, shadow_n_open_rt);
end

if ( err_mk ) begin
   n_errs_mk++;
   if ( n_errs_mk < n_errors_show ) begin
      $write("Error %s: '%s' != '%s' (correct)\n",
             prefix_txt,
             char_to_string(str_marked), char_to_string(shadow_str_e));
   end
```

```
            end

        end

        $write("Done with tests %s. Errors: %0d cl, %0d op, %0d mark.\n",
               prefix_txt,
               n_errs_cl, n_errs_op, n_errs_mk);

        testbench.t_errs_each_cl[mtype][n] = n_errs_cl;
        testbench.t_errs_each_op[mtype][n] = n_errs_op;
        testbench.t_errs_each_mk[mtype][n] = n_errs_mk;

        done = 1;

    end

endmodule

// cadence translate_on
```

LSU EE 4755 Homework 4 Solution Due: 8 Nov 2024

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 the students' responsibility 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, 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**.

Problems start on next page.

Problem 1: Appearing below is the base case from module `pmatch_mark` in the solution to Homework 3 Problem 2.

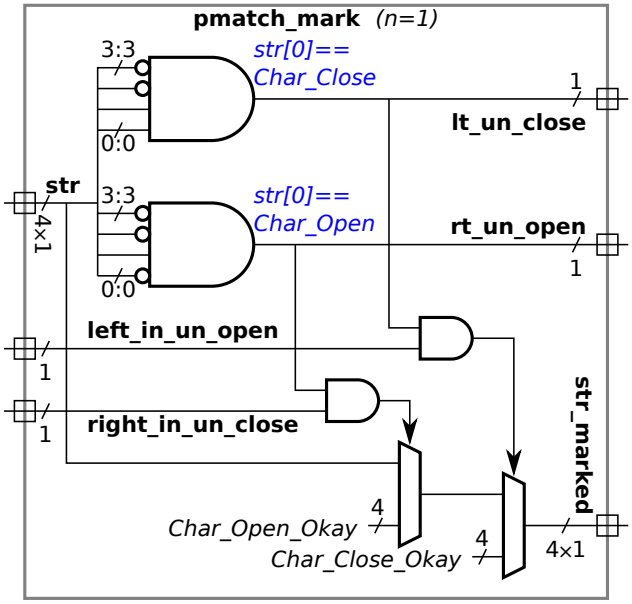
```
typedef enum logic [3:0]
  { Char_Blank = 0,     Char_Dot = 1,
    Char_Open = 2,      Char_Close = 3,
    Char_Open_Okay = 4, Char_Close_Okay = 5 } Char;

module pmatch_mark
  #( int n = 5, wn = $clog2(n+1) )
   ( output logic [wn-1:0] left_out_n_unmat_close, right_out_n_unmat_open,
     output uwire [3:0] str_marked [0:n-1],
     input uwire [wn-1:0] left_in_n_unmat_open, right_in_n_unmat_close,
     input uwire [3:0] str [0:n-1] );

   if ( n == 1 ) begin

      assign left_out_n_unmat_close = str[0] == Char_Close;
      assign right_out_n_unmat_open = str[0] == Char_Open;

      assign str_marked[0] =
             str[0] == Char_Close && left_in_n_unmat_open   ? Char_Close_Okay :
             str[0] == Char_Open  && right_in_n_unmat_close ? Char_Open_Okay  :
             str[0];
```

☑ Show the hardware that will be inferred for the base case (`n==1`) shown above.

☑ Show the hardware after optimization and ☑ for the default value of `wn`.

☑ In the optimized hardware do not show comparison units, instead show the individual gates performing the comparison, ☑ optimizing for constant values.

Solution appears to the right with some abbreviated connection names. As in the midterm exam the equality comparisons have been optimized into AND gates. Notice that because $n = 1$ input `str` has a single 4-bit element and so `str[0]` refers to all of `str` which is four bits and carries a single `Char`. Because the default value of `wn` is to be used and `n=1` is given we know that $w_n = \lceil \lg(n+1) \rceil = \lceil \lg(2) \rceil = 1$. That means the `left_in_un_open` (a.k.a. `left_in_n_unmat_open`) is one bit and so it can be directly connected to an AND gate input. If $w_n > 1$ then an OR gate would be needed before the AND gate. *Final exam short answer question?*

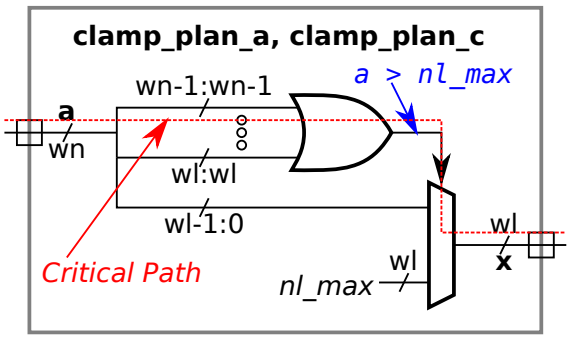


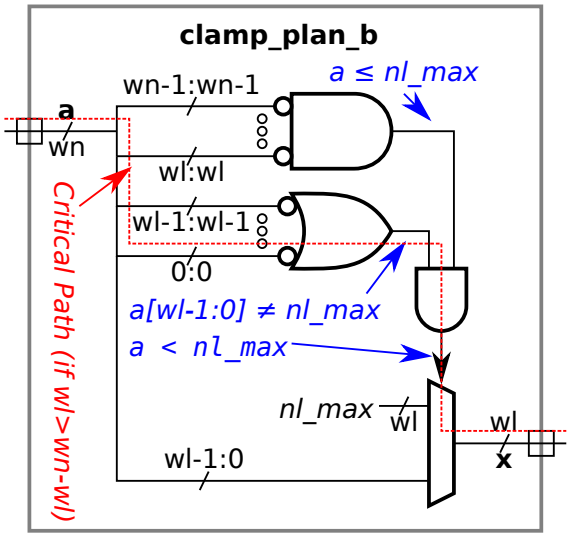
Problem 2: Appearing below are three variations on a module that will set its output to either the input value, or a maximum value if the input is larger. The module will always be instantiated with `wl < wn`. All of them are functionally equivalent, but were synthesized to different costs (by Genus 23.12-s086.1 when similar code was used in the solution to Homework 3). Because they are functionally equivalent a perfect synthesis program would synthesize each to the same hardware (with equal costs).

```
module clamp_plan_a
  #( int wl = 3, wn = 4 ) ( output uwire [wl-1:0] x, input uwire [wn-1:0] a );
   localparam logic [wl-1:0] nl_max = ~(wl)'(0); // Sequence of wl 1s.
   assign x = a <= nl_max ? a : nl_max;
endmodule

module clamp_plan_b
  #( int wl = 3, wn = 4 ) ( output uwire [wl-1:0] x, input uwire [wn-1:0] a );
   localparam logic [wl-1:0] nl_max = ~(wl)'(0); // Sequence of wl 1s.
   assign x = a < nl_max ? a : nl_max;
endmodule

module clamp_plan_c
  #( int wl = 3, wn = 4 ) ( output uwire [wl-1:0] x, input uwire [wn-1:0] a );
   localparam logic [wl-1:0] nl_max = ~(wl)'(0); // Sequence of wl 1s.
   assign x = !a[wn-1:wl] ? a : nl_max;
endmodule
```

☑ Show the optimized hardware for the low-cost version(s).

Solution appears to the right. Because `nl_max` is w_l bits and $w_l > w_n$ it is possible to determine that `a > nl_max` by checking if any of the bits in positions `wn-1:wl` are 1. If so, `a > nl_max`, otherwise `a <= nl_max`. The Plan C hardware explicitly uses this optimization, and is shown to the right. For Plan A the solution is based on the assumption that the synthesis program figures out the optimization. The hardware is more complicated for Plan B because to compute `a<nl_max` the hardware needs to compute `a<=nl_max` (the AND gate) and `a!=nl_max` (the OR gate). The Plan B solution is based on a synthesis program that can optimize the comparison this way, but one that can't figure out that there was no need to check for the `a!=nl_max` case because if `a==nl_max` either multiplexor input can be used.





Solution continued on next page.

☑ Find the simple-model cost of each after optimization. The costs should be ☑ in terms of `wn` and `wl`.

Plan A and Plan C: Multiplexor (one constant input): $w_l\,\mathrm{u_c}$, OR gate $[w_n - w_l - 1]\,\mathrm{u_c}$.

Plan B: Multiplexor (one constant input): $w_l\,\mathrm{u_c}$, "big" AND gate $[w_n - w_l - 1]\,\mathrm{u_c}$, OR gate $w_l\,\mathrm{u_c}$, little AND gate $1\,\mathrm{u_c}$. Note that the big AND gate might just have one input so not only would it not be big, but it would not be an AND gate at all, just a NOT gate.

☑ Find the simple-model delay of each after optimization. The delays should be ☑ in terms of `wn` and `wl`.

Plan A and Plan C: Delay of OR gate, $\lceil \lg(w_n - w_l) \rceil\,\mathrm{u_t}$. Delay of mux, $1\,\mathrm{u_t}$. Critical path, shown in red on the diagram, is through both so the critical path delay is $[\lceil \lg(w_n - w_l) \rceil + 1]\,\mathrm{u_t}$.

Plan B: Delay of "big" AND gate, $\lceil \lg(w_n - w_l) \rceil\,\mathrm{u_t}$. Delay of OR gate $\lceil \lg(w_l) \rceil\,\mathrm{u_t}$. Delay of AND gate $1\,\mathrm{u_t}$, delay of mux, $1\,\mathrm{u_t}$. Assuming $w_l > w_n - w_l$ the critical path, shown on red in the diagram, is through the OR gate, little AND gate and mux for a total delay of $[\lceil \lg(w_l) \rceil + 1 + 1]\,\mathrm{u_t}$. Notice that the critical path length is much larger in Plan B under the reasonable assumption that $w_l > w_n - w_l$.

Problem 3: Appearing on the next page is a simplified solution to Homework 3, Problem 2. In this module the number of bits in the connections carrying parentheses counts is hardcoded to 8. Though the hardware is correct for $n < 256$ it is more costly and slower than it needs to be. But for this problem it's good enough.

Show the Homework 3, Problem 2 hardware that will be inferred for this module for $n > 1$ (the non-base case). That is, don't show the hardware computing `left_out_n_unmat_close` and `right_out_n_unmat_open`.

☑ Show the inferred hardware at one level for $n > 1$.

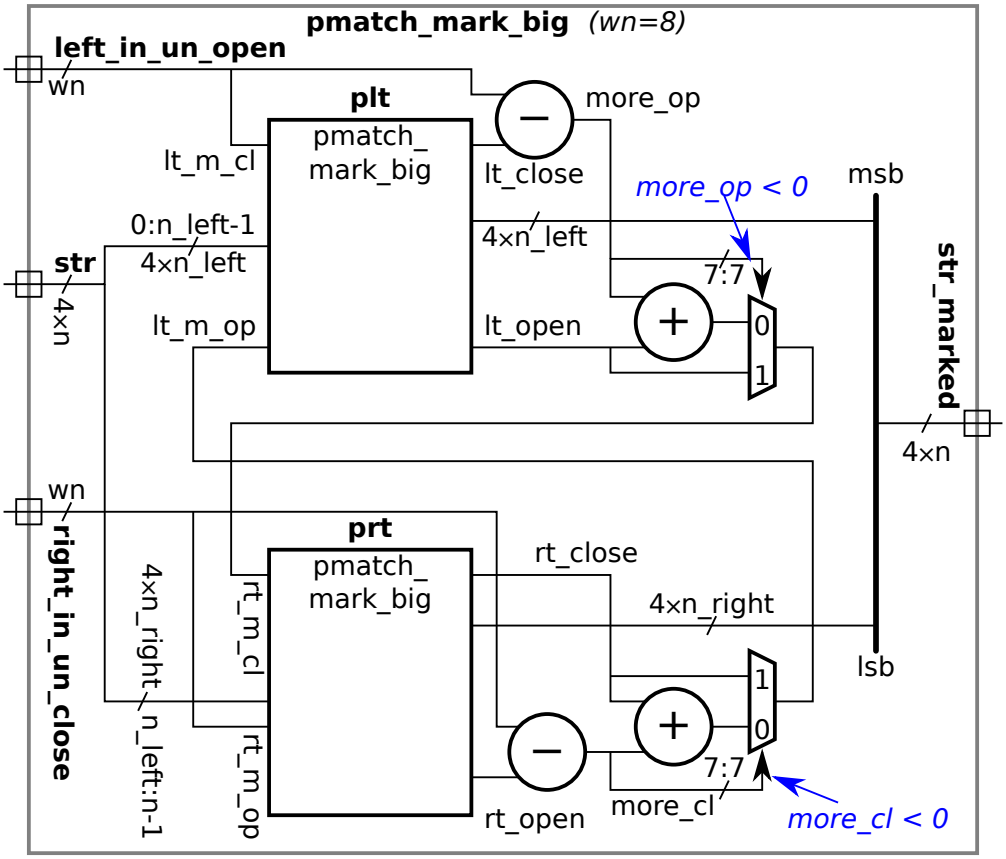
☑ Feel free to use abbreviations.

☑ Don't show the Homework 3 Problem 1 hardware (the last `always_comb`).

☑ Don't confuse elaboration-time computation with hardware.

Solution appears below. Note that the condition `more_op<0` can be evaluated by just looking at the most-significant bit, 7 in this case.

To help understand the circuit try tracing the critical path. To do that consider two cases: the root (the top-level instantiation of `pmatch_mark_big`) and a lower-level instantiation (but not the base case). In the root assume that `left_in_un_open` and `right_in_un_close` are available at $t = 0$.



```
module pmatch_mark_big #( int n = 5, wn = 8 )
   ( output logic [wn-1:0] left_out_n_unmat_close, right_out_n_unmat_open,
     output uwire [3:0] str_marked [0:n-1],
     input uwire [wn-1:0] left_in_n_unmat_open, right_in_n_unmat_close,
     input uwire [3:0] str [0:n-1] );

   if ( n == 1 ) begin
      assign left_out_n_unmat_close = str[0] == Char_Close;
      assign right_out_n_unmat_open = str[0] == Char_Open;
      assign str_marked[0] =
             str[0] == Char_Close && left_in_n_unmat_open   ? Char_Close_Okay :
             str[0] == Char_Open  && right_in_n_unmat_close ? Char_Open_Okay  : str[0];
   end else begin

      localparam int n_left = n/2,   n_right = n - n_left;
      localparam int wl = 8,         wr = 8;  // Note: this is wasteful.

      uwire [wl-1:0] lt_close, lt_open;
      uwire [wr-1:0] rt_close, rt_open;
      logic [wl-1:0] lt_matched_op, lt_matched_cl;
      logic [wr-1:0] rt_matched_op, rt_matched_cl;

      pmatch_mark_big #(n_left,wl) plt   // Recursive Instantiation
        ( lt_close, lt_open, str_marked[0:n_left-1],
          lt_matched_cl, lt_matched_op, str[0:n_left-1] );
      pmatch_mark_big #(n_right,wr) prt  // Recursive Instantiation
        ( rt_close, rt_open, str_marked[n_left:n-1],
          rt_matched_cl, rt_matched_op, str[n_left:n-1] );

      always_comb begin
         logic signed [wn-1:0] more_op, more_cl;

         lt_matched_cl = left_in_n_unmat_open;
         rt_matched_op = right_in_n_unmat_close;

         more_op = left_in_n_unmat_open - lt_close;
         rt_matched_cl = more_op < 0 ? lt_open : more_op + lt_open;

         more_cl = right_in_n_unmat_close - rt_open;
         lt_matched_op = more_cl < 0 ? rt_close : more_cl + rt_close;
      end

      always_comb begin          // Same as Homework 3 Problem 1
         logic signed [wn-1:0] delta;
         delta = lt_open - rt_close;
         left_out_n_unmat_close = delta >= 0 ? lt_close : lt_close - delta;
         right_out_n_unmat_open = delta < 0  ? rt_open  : rt_open + delta;
      end
   end
endmodule
```

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2024 Homework 5 -- SOLUTION
//

 /// Assignment https://www.ece.lsu.edu/koppel/v/2024/hw05.pdf

`default_nettype none


////////////////////////////////////////////////////////////////////////////////
/// Problem 1
//
  /// Complete dot_seq_2_base.
//
//     [✔] The module can use procedural code.
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//
//     [✔] Don't assume any particular parameter values.
//
//     [✔] As always, code must be written clearly.
//     [✔] As always, pay attention to cost and performance.

module dot_seq_2
  #( int w = 5, wi = 4 )
   ( output logic [w-1:0] dp,
     output logic [wi-1:0] first_id, last_id,
     input uwire [w-1:0] a[2], b[2],
     input uwire [wi-1:0] in_id,
     input uwire reset, first, last,
     input uwire clk );

   // [✔] Critical path can't include more than one arithmetic operation.
   // [✔] Don't change outputs until operation is complete.

   /// SOLUTION

   /// Declare Pipeline Latch Registers
   //
   // A pipeline latch is an edge-triggered register that carries
   // values passing from stage to stage.
   //
   // Index values in the first unpacked dimension indicate the
   // stage(s) for which a pipeline latch is declared. For example,
   // pl_a is only declared for stage 1, pl_prod is only declared for
   // stage 2, but pl_id declares three registers, pl_id[1] for stage
   // 1, pl_id[2] for stage 2, and pl_id[3] for stage 3. Note that
   // pl_a[1] is a stage-1 register that holds two w-bit values. A
   // register declared for stage x is written by hardware in stage
   // x-1 and read by hardware in stage x.
   //
```

```
logic [w-1:0] pl_a[1:1][2], pl_b[1:1][2];  // Arriving vector elements.
logic [w-1:0] pl_prod[2:2][2];             // Vector products.
logic [w-1:0] pl_sum[3:3];                 // Dot prod of 2-element segment.
logic [wi-1:0] pl_id[1:3];                 // ID.
logic [1:0] pl_fl[1:3];                    // The first and last signals.
//
// Note that ID, first, and last, move through the pipeline
// unchanged, which is why there is a pipeline latch in each stage
// to carry their values. In contrast, vector elements (arriving in
// a and b) are transformed in each stage and so the names of the
// pipeline latches carrying their values and values computed from
// them change at each stage: a/b -> prod -> sum.

/// Declare Accumulator Registers
//
// An accumulator holds values that don't move with the pipeline.
// Here the accumulators are updated by values from stage 3.
//
logic [wi-1:0] acc_id;
logic [w-1:0] acc_sum;

always_ff @( posedge clk ) begin

   /// Stage 0
   //
   // Move arriving inputs into pipeline latches. Since the problem
   // states inputs arrive late in the cycle it is not possible to
   // do any calculations using their values until the next cycle.
   //
   pl_a[1] <= a;  // This copies both elements of a.
   pl_b[1] <= b;
   pl_id[1] <= in_id;
   pl_fl[1] <= reset ? 2'b0 : {last,first};

   /// Stage 1
   //
   // Compute products ..
   //
   for ( int i=0; i<2; i++ ) pl_prod[2][i] <= pl_a[1][i] * pl_b[1][i];
   //
   // .. and move everyone else along unchanged (except for reset).
   //
   pl_id[2] <= pl_id[1];
   pl_fl[2] <= reset ? 2'd0 : pl_fl[1];

   /// Stage 2
   //
   // Compute sum ..
   //
   pl_sum[3] <= pl_prod[2][0] + pl_prod[2][1];
   //
   // .. and move everyone else along unchanged (except for reset).
   //
   pl_id[3] <= pl_id[2];
```

```
   pl_fl[3] <= reset ? 2'h0 : pl_fl[2];

   /// Stage 3
   //
   begin
      // Declare intermediate values in this block.
      //
      automatic logic s3_first = pl_fl[3][0]; // For readability.
      automatic logic s3_last =  pl_fl[3][1]; // For readability.

      // Add arriving value on to accumulated sum, unless this
      // is the first set of vector elements.
      //
      automatic logic [w-1:0] s3_sum =
        s3_first ? pl_sum[3] : pl_sum[3] + acc_sum;
      //
      // Write the sum to the accumulator register.
      //
      acc_sum <= s3_sum;
      //
      // Note that there is no need to check reset or last because
      // if they were true the value written to acc_sum would not
      // be used and so there is no need to waste hardware that
      // would avoid writing it.

      if ( reset ) begin

         // Set output IDs to zero on a reset.
         //
         first_id <= 0;
         last_id <= 0;
         //
         // Since the problem does not say to set dp to zero there
         // is no need to waste hardware doing so.

      end else begin

         // Hold on to the ID if this is the beginning of the
         // vector.
         //
         if ( s3_first ) acc_id <= pl_id[3];
         //
         // A beginner's mistake is to use the value of first
         // currently at the module inputs rather than the value
         // in pipeline latch pl_fs[3].

         if ( s3_last ) begin

            // If this is the end of the vector, update the module
            // outputs.
            //
            first_id <= s3_first ? pl_id[3] : acc_id;
            last_id <= pl_id[3];
            dp <= s3_sum;
```

```
            end
          end
        end

    end

endmodule

`ifdef xxxx
Synthesizing at effort level "high"

Module Name                              Area    Delay    Delay     Synth
                                                Actual   Target     Time
dot_seq_2_w8_wi8                       148802    3.58   900.0 ns      8 s
dot_seq_2_w8_wi8_17                    219405    2.02     0.1 ns     43 s
`endif


////////////////////////////////////////////////////////////////////////////
/// Testbench Code
//
// It is okay to modify the testbench code to facilitate the coding
// and debugging of your modules. Keep in mind that your submission
// will be tested using a different testbench, so on the one hand no
// one will be accused of dishonesty for modifying the testbench
// below. However be sure to restore any changes to make sure that
// your code passes the original testbench.


// cadence translate_off

program reactivate
    (output uwire clk_reactive, output int cycle_reactive,
     input uwire clk, input var int cycle);
    assign clk_reactive = clk;
    assign cycle_reactive = cycle;
endprogram


module dot_seq_m
  #( int n = 1, w = 5, wi = 4 )
    ( output logic [w-1:0] dp,
      output logic [wi-1:0] first_id, last_id,
      input uwire [w-1:0] a[n], b[n],
      input uwire [wi-1:0] in_id,
      input uwire reset, first, last,
      input uwire clk );

    if ( n == 1 ) begin

       dot_seq #(w,wi) d(dp,first_id,last_id,a[0],b[0],in_id,reset,first,last,clk);

    end else if ( n == 2 ) begin
```

```
      dot_seq_2 #(w,wi) d(dp,first_id,last_id,a,b,in_id,reset,first,last,clk);

   end

endmodule

module testbench;

   logic done;

   localparam int m = 2;
   localparam int n_tests = 10000;
   localparam int wi = 8;
   localparam int w = 8;

   // Maximum number of cycles from "last" signal to arrival of outputs.
   localparam int latency_max = 5;
   // Minimum latency that will be considered correct.
   localparam int latency_min = 2 + $clog2(m);

   localparam int cyc_max = n_tests * 1000;

   int seed;
   initial seed = 475501;

   function automatic bit rand_bern( int period );
      rand_bern = $dist_uniform(seed,1,period) == 1;
   endfunction

   bit clk;
   int cycle, cycle_limit;
   logic clk_reactive;
   int cycle_reactive;
   reactivate ra(clk_reactive,cycle_reactive,clk,cycle);
   string event_trace;
   string ev_trace[$];

   initial begin
      clk = 0;
      cycle = 0;
      event_trace = "";

      done = 0;
      cycle_limit = cyc_max;
      //  wait( tstart );

      fork
         while ( !done ) #1 cycle += clk++;
         wait( cycle >= cycle_limit )
           $write("Exit from clock loop at cycle %0d, limit %0d.  %s\n %s\n",
                  cycle, cycle_limit, "** CYCLE LIMIT EXCEEDED **",
                  event_trace);
      join_any;
```

```
   done = 1;
end

typedef struct
  {
   logic [w-1:0] dp;
   logic [wi-1:0] first_id, last_id;
   int cyc; // Cycle that last input asserted.
   int latency; // Number of cycles from last to dp to appearing at output.
   bit correct;
   } Info;

uwire [w-1:0] dp;
uwire [wi-1:0] out_first_id, out_last_id;
logic [w-1:0] a[m], b[m];
logic [wi-1:0] in_id;
logic reset, first, last;
bit done_tests, start_check;

dot_seq_m #(m,w,wi)
ds(dp,out_first_id,out_last_id,a,b,in_id,reset,first,last,clk);

enum { S_reset, S_gap, S_continue } State;
Info exp_info[$];
int n_tests_completed;
Info info_null;

initial begin

   automatic int state = S_reset;
   logic [w-1:0] shadow_sum;
   logic [wi-1:0] shadow_id;

   n_tests_completed = 0;
   info_null.first_id = 0;
   info_null.last_id = 0;
   info_null.dp = 0;
   info_null.cyc = cycle;
   info_null.correct = 0;

   exp_info.push_back(info_null);

   start_check = 0;
   done_tests = 0;
   first = 0;
   last = 0;
   for ( int j=0; j<m; j++ ) begin a[j]=0; b[j]=0; end
   in_id = $dist_uniform(seed,1,(1<<wi)-1);
   reset = 1;
   @( negedge clk ); @( negedge clk );
   start_check = 1;

   while ( n_tests_completed < n_tests ) begin
```

```
string tr_entry;

first = 0;
last = 0;
reset = 0;
for ( int j=0; j<m; j++ ) begin
   if ( n_tests_completed < 20 ) begin
      a[j] = 1;          b[j] = 1 << ( j * 4 );
   end else begin
      a[j] = {$random};  b[j]= {$random};
   end
end

// For error detection recent input IDs must be distinct.
in_id += $dist_uniform(seed,1,3);
if ( in_id == 0 ) in_id = 1;

if ( state == S_reset ) state = S_gap;
if ( rand_bern(100) ) begin
   reset = 1;
   in_id = $dist_uniform(seed,1,(1<<wi)-1);
   exp_info.delete();
   info_null.cyc = cycle;
   exp_info.push_back(info_null);
   state = S_reset;
end

if ( state == S_gap ) begin
   first = rand_bern(3);
   if ( first ) begin
      state = S_continue;
      shadow_sum = 0;
      shadow_id = in_id;
   end
end

for ( int j=0; j<m; j++ )
  shadow_sum += a[j] * b[j];

if ( state == S_continue ) begin
   last = rand_bern(4);
   if ( last ) begin
      Info info;
      state = S_gap;
      n_tests_completed++;
      info.dp = shadow_sum;
      info.first_id = shadow_id;
      info.last_id = in_id;
      info.cyc = cycle;
      info.correct = 0;
      exp_info.push_back(info);
   end
end
```

```
      if ( exp_info.size() > 1 && exp_info[1].cyc + latency_max < cycle )
        void'(exp_info.pop_front());

      @( negedge clk );

    end

    first = 0;
    last = 0;

    done_tests = 1;

    $write("Done with inputs.\n");

    begin
      automatic int cyc_limit = cycle + 20;
      wait ( cycle == cyc_limit );
      done = 1;

    end

  end

  initial begin

    automatic int sum_latency = 0;
    automatic int n_correct = 0;
    automatic int n_err_fid = 0;
    automatic int n_err_lid = 0;
    automatic int n_err_sum = 0;
    automatic int n_err_time = 0;
    automatic int n_err_early = 0;
    automatic int err_fid_cyc = 0;
    automatic int err_lid_cyc = 0;
    automatic int err_dp_cyc = 0;
    string tr_entry;

    #0;

    wait( start_check );

    while ( !done ) begin
      Info info, info_dp;
      automatic string err_text[$];
      string tr_entry;
      int fid_idx, lid_idx, dp_idx, age;
      bit err_time, err_fid, err_lid, err_early;

      @( posedge clk_reactive );

      if ( done ) break;

      fid_idx = -1;  lid_idx = -1;  dp_idx = -1;
```

```
info_dp = info_null; age = -1;
err_early = 0;

for ( int i=0; i<exp_info.size(); i++ ) begin
   automatic Info info = exp_info[i];
   automatic int bi = exp_info.size() -1 -i;
   automatic int nm = 0;
   if ( info.first_id === out_first_id ) begin fid_idx = bi; nm++; end
   if ( info.last_id === out_last_id ) begin lid_idx = bi; nm++; end
   if ( info.dp === dp ||
        info.first_id == 0 && out_first_id === 0 &&
        info.last_id == 0 && out_last_id === 0 )
     begin
        dp_idx = bi; nm++;
        if ( !info.correct && info.first_id ) begin
           automatic int latency = cycle - info.cyc;
           exp_info[i].correct = 1;
           exp_info[i].latency = latency;
           sum_latency += latency;
           err_early = latency < 3;
           n_correct++;
        end
        info_dp = exp_info[i];
     end
   if ( nm == 3 ) break;
end

info = exp_info[$];

if ( dp_idx >= 0 ) age = cycle - info_dp.cyc;

err_fid = fid_idx == -1 || dp_idx >= 0 && fid_idx != dp_idx;
err_lid = lid_idx == -1 || dp_idx >= 0 && lid_idx != dp_idx;

if ( err_fid && info.cyc != err_fid_cyc ) begin
   n_err_fid++;
   err_fid_cyc = info.cyc;
   if ( n_err_fid < 4 )
     err_text.push_back
       ( $sformatf("Error first ID: %0h != %0h (correct)\n",
                   out_first_id, info_dp.first_id ) );
end
if ( err_lid && info.cyc != err_lid_cyc ) begin
   n_err_lid++;
   err_lid_cyc = info.cyc;
   if ( n_err_lid < 4 )
     err_text.push_back
       ( $sformatf("Error  last ID: %0h != %0h (correct)\n",
                   out_last_id, info_dp.last_id ) );
end

if ( err_early ) begin
   n_err_early++;
   if ( n_err_early < 4 )
```

```
      err_text.push_back
        ( $sformatf("Error dp timing: latency %0d cyc < 3 cyc (minimum)\n",
                    info_dp.latency) );
end

if ( dp_idx == -1 && info.cyc != err_dp_cyc ) begin
   n_err_sum++;
   err_dp_cyc = info.cyc;
   if ( n_err_sum < 4 )
     err_text.push_back
     ( $sformatf("Error dp: %h != %h (correct)\n", dp, info.dp ) );
end

err_time = 0 && fid_idx >= 0 && lid_idx >= 0 && dp_idx >= 0
  && ( fid_idx != lid_idx || lid_idx != dp_idx );

if ( err_time ) begin
   n_err_time++;
   if ( n_err_time < 4 )
     err_text.push_back
     ( $sformatf("Error timing: %0d %0d %0d (should all be same)\n",
           fid_idx, lid_idx, dp_idx) );
end

begin
   automatic string a_str, b_str;
   for ( int j=0; j<m; j++ ) begin
      a_str = { a_str, j?",":"", $sformatf("%h",a[j]) };
      b_str = { b_str, j?",":"", $sformatf("%h",b[j]) };
   end

tr_entry =
  { $sformatf("%4d %s%s%s ID %h  A %s  B %s",
              cycle,
              reset ? "R" : "_",
              first ? "F" : "_",
              last ? "L" : "_",
              in_id, a_str, b_str ),
    "  ",
    $sformatf("exp: fid %h  lid %h  dp %h",
              exp_info[$].first_id, exp_info[$].last_id,
              exp_info[$].dp),
    "  ",
    $sformatf("MOD: %s %h  %s %h  %s %h",
              err_fid ? "ERR" : err_time ? "err" : "FID",
              out_first_id,
              err_lid ? "ERR" : err_time ? "err" : "LID",
              out_last_id,
              dp_idx == -1 ? "ER" : age < 3 ? "EY" : err_time ? "er" : "DP",
              dp )
    };
end

ev_trace.push_back(tr_entry);
```

```
        while ( ev_trace.size() > 10 ) tr_entry = ev_trace.pop_front();

        if ( err_text.size() || cycle < 20 ) begin

           while ( ev_trace.size() ) $write("%s\n",ev_trace.pop_front());

           //  foreach ( ev_trace[i] ) $write("%s\n",ev_trace[i]);
           foreach ( err_text[i] ) $write("%s",err_text[i]);

        end

     end

     begin

        automatic real avg_latency =
          n_correct ? sum_latency / real'(n_correct) : 0;

        $write("Done with %0d tests. %0d dp errs ( %0d correct )\n",
               n_tests_completed, n_err_sum, n_correct);

        //  $write("Done with %0d tests. %0d ID errs,  %0d FID errs, %0d LID errs.\n",
        $write("Done with %0d tests. %0d FID errs, %0d LID errs.\n",
               n_tests_completed, n_err_fid, n_err_lid);
        $write("Done with %0d tests. Correct %0d, avg latency %.1f cyc %0s\n",
               n_tests_completed, n_correct, avg_latency,
               avg_latency < latency_min ?
               $sformatf("Error, too low, %0d cyc minimum.",latency_min) : "Okay");

     end

  end


endmodule


// cadence translate_on
```

LSU EE 4755 Homework 6 Solution Due: 1 Dec 2024

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, 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**.

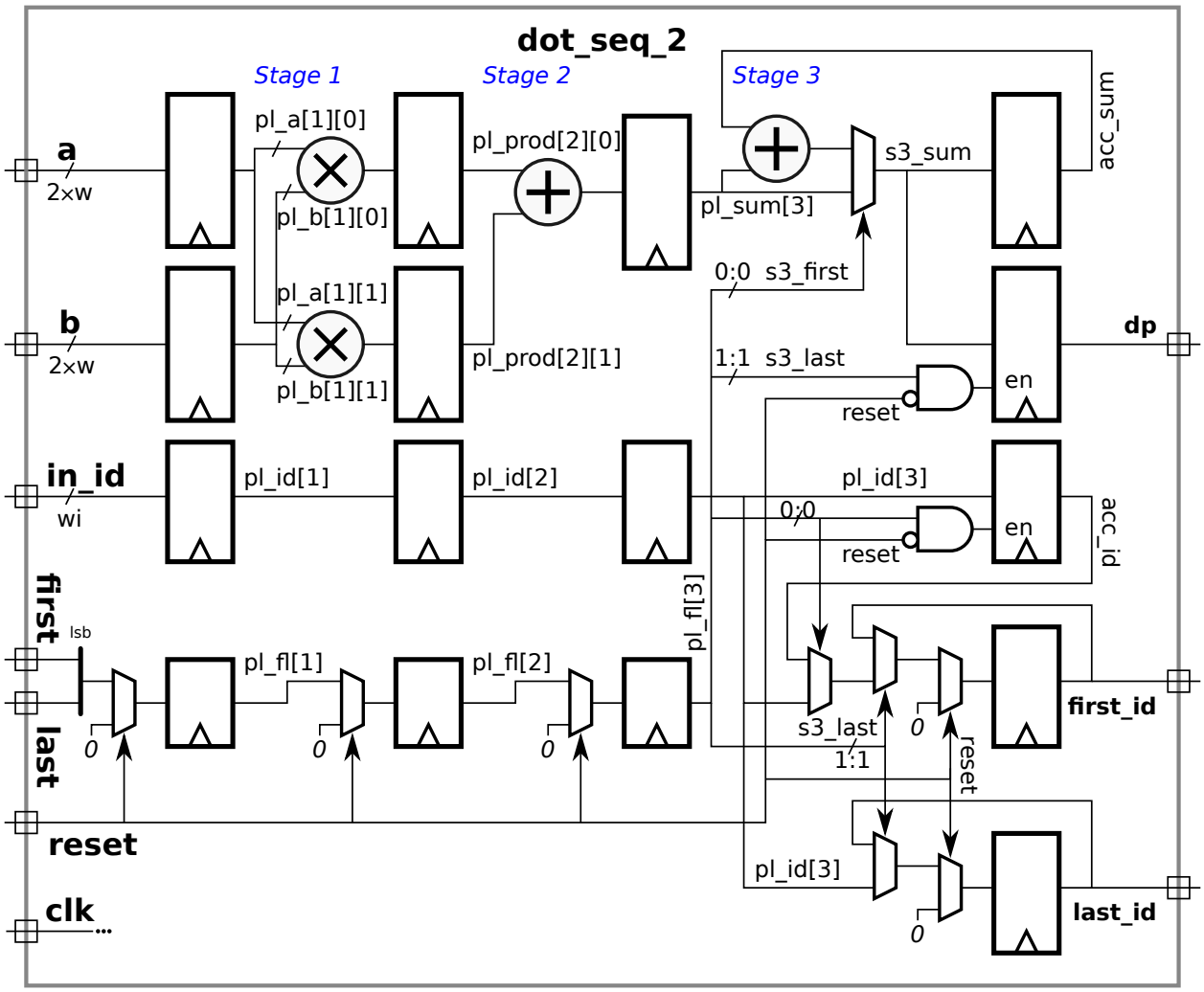
Problems start on next page.

Problem 1: Show the hardware that will be synthesized for the posted solution to Homework 5. The solution (with fewer comments than the posted version) is shown below.

Solution appears on the page after the module.

```
module dot_seq_2 #( int w = 5, wi = 4 )
   ( output logic [w-1:0] dp,          output logic [wi-1:0] first_id, last_id,
     input uwire [w-1:0] a[2], b[2],   input uwire [wi-1:0] in_id,
     input uwire reset, first, last,   input uwire clk );

   logic [w-1:0] pl_a[1:1][2], pl_b[1:1][2];  // Arriving vector elements.
   logic [w-1:0] pl_prod[2:2][2];             // Vector products.
   logic [w-1:0] pl_sum[3:3];                 // Dot prod of 2-element segment.
   logic [wi-1:0] pl_id[1:3];                 // ID.
   logic [1:0] pl_fl[1:3];                    // The first and last signals.
   logic [wi-1:0] acc_id;
   logic [w-1:0] acc_sum;

   always_ff @( posedge clk ) begin
      /// Stage 0
      pl_a[1] <= a;  // This copies both elements of a.
      pl_b[1] <= b;
      pl_id[1] <= in_id;
      pl_fl[1] <= reset ? 2'b0 : {last,first};

      /// Stage 1
      for ( int i=0; i<2; i++ ) pl_prod[2][i] <= pl_a[1][i] * pl_b[1][i];
      pl_id[2] <= pl_id[1];
      pl_fl[2] <= reset ? 2'd0 : pl_fl[1];

      /// Stage 2
      pl_sum[3] <= pl_prod[2][0] + pl_prod[2][1];
      pl_id[3] <= pl_id[2];
      pl_fl[3] <= reset ? 2'h0 : pl_fl[2];

      /// Stage 3
      begin
         automatic logic s3_first = pl_fl[3][0]; // For readability.
         automatic logic s3_last =  pl_fl[3][1]; // For readability.
         automatic logic [w-1:0] s3_sum = s3_first ? pl_sum[3] : pl_sum[3] + acc_sum;

         acc_sum <= s3_sum;

         if ( reset ) begin
            first_id <= 0;
            last_id <= 0;
         end else begin
            if ( s3_first ) acc_id <= pl_id[3];
            if ( s3_last ) begin
               first_id <= s3_first ? pl_id[3] : acc_id;
               last_id <= pl_id[3];
               dp <= s3_sum;
            end
         end
      end
   end
endmodule
```

Solution to Problem 1 appears below.



Problem 2: Solve 2023 Final Exam Problem 2, which asks for a cost and delay analysis of a word count module.

See the posted final exam solution.

15 Fall 2023 Solutions

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2023 Homework 1 -- SOLUTION
//

 /// Assignment https://www.ece.lsu.edu/koppel/v/2023/hw01.pdf

`default_nettype none

////////////////////////////////////////////////////////////////////////////////
/// Problem 1
//
  /// Complete minmax2p1 using a compare_lt and mux2 instantiations.
 ///
//
//     [✔] Only modify minmax2p1. Use minmax2 for reference.
//
//     [✔] minmax2p1 must instantiate a compare_lt module and mux2 modules.
//     [✔] minmax2p1 must NOT use assign statements or procedural code.
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//
//     [✔] Don't assume any particular parameter values.
//
//     [✔] Code must be written clearly.

module minmax2p1
  #( int w = 4 )
   ( output uwire [w-1:0] min, max,
     input uwire [w-1:0] a0, a1 );

   // Put solution here.

   /// SOLUTION
   uwire lt;
   compare_lt #(w) clt(lt, a0, a1);

   mux2 #(w) mn(max,lt,a0,a1);
   mux2 #(w) mx(min,lt,a1,a0);

endmodule

module compare_lt
  #( int w = 31 )
   ( output uwire lt,
     input uwire [w-1:0] a0, a1 );

   // DO NOT modify this module.

   // Set lt to 0 if a1 < a0, set lt to 1 otherwise.
   //
```

```
   assign lt = a0 <= a1;

endmodule

module mux2
  #( int w = 3 )
   ( output uwire [w-1:0] x,
     input uwire s,
     input uwire [w-1:0] a0, a1 );

   // DO NOT modify this module either.

   assign x = s ? a1 : a0;

endmodule

module minmax2
  #( int w = 10 )
   ( output uwire [w-1:0] min, max,
     input uwire [w-1:0] a0, a1 );

   // DO NOT modify this module either.

   // Assign min to the smaller of a0 and a1, and max to the larger.
   assign { min, max } = a0 <= a1 ? { a0, a1 } : { a1, a0 };

endmodule



////////////////////////////////////////////////////////////////////////////////
/// Problem 2
//
  /// Complete minmax4 and minmax8.
//
//     [✔] In minmax4, instantiate minmax2.
//     [✔] In minmax8, instantiate minmax4.
//     [✔] In minmax4 and minmax8, instantiate min2 and max2, as necessary.
//
//     [✔] Do not use assign statements or procedural code.
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//
//     [✔] Don't assume any particular parameter values.
//
//     [✔] Pay attention to cost.
//     [✔] Assume cost of min2 + max2 in more than minmax2.
//     [✔] Code must be written clearly.

module minmax4
  #( int w = 20 )
   ( output uwire [w-1:0] min, max,
     input uwire [w-1:0] a[4] );
```

```
   // Put solution here.

   /// SOLUTION

   uwire [w-1:0] lomin, lomax, himin, himax;

   minmax2 #(w) mlo( lomin, lomax, a[0], a[1] );
   minmax2 #(w) mhi( himin, himax, a[2], a[3] );

   min2 #(w) m1( min, lomin, himin );
   max2 #(w) m2( max, lomax, himax );

endmodule


module minmax8
  #( int w = 12 )
   ( output uwire [w-1:0] min, max,
     input uwire [w-1:0] a[8] );

   // Put solution here.

   /// SOLUTION

   uwire [w-1:0] lomin, lomax, himin, himax;

   minmax4 #(w) mlo( lomin, lomax, a[0:3] );
   minmax4 #(w) mhi( himin, himax, a[4:7] );

   min2 #(w) m1( min, lomin, himin );
   max2 #(w) m2( max, lomax, himax );

endmodule


module min2
  #( int w = 10 )
   ( output uwire [w-1:0] min,
     input uwire [w-1:0] a0, a1 );
   assign min = a0 < a1 ? a0 : a1;
endmodule

module max2
  #( int w = 10 )
   ( output uwire [w-1:0] max,
     input uwire [w-1:0] a0, a1 );
   assign max = a0 < a1 ? a1 : a0;
endmodule



////////////////////////////////////////////////////////////////////////////////
/// Testbench Code
```

```
// cadence translate_off

module testbench;

   localparam int npsets = 3; // Number of instantiations.
   localparam int pset[npsets] =
              '{ 2, 4, 8 };

   int t_errs;       // Total number of errors.
   initial begin t_errs = 0; end
   final $write("Total number of errors: %0d\n",t_errs);

   uwire d[npsets:-1];    // Start / Done signals.
   assign d[-1] = 1;  // Initialize first at true.

   // Instantiate a testbench at each size.
   //
   for ( genvar i=0; i<npsets; i++ )
     testbench_n #(pset[i]) t2( .done(d[i]), .tstart(d[i-1]) );

endmodule

module testbench_n
  #( int n = 5 )
   ( output logic done, input uwire tstart );

   localparam int w = 13;
   localparam int ntests = 100;

   logic [w-1:0] a[n], sa[n];
   uwire [w-1:0] min, max;

   if ( n == 2 )
     minmax2p1 #(w) mm( min, max, a[0], a[1] );
   else if ( n == 4 )
     minmax4 #(w) mm( min, max, a );
   else if ( n == 8 )
     minmax8 #(w) mm( min, max, a );

   int n_err_min, n_err_max;

   initial begin

      done = 0;
      wait( tstart );

      n_err_min = 0;
      n_err_max = 0;

      for ( int i=0; i<ntests; i++ ) begin

         logic [w-1:0] shadow_min, shadow_max;
```

```
      for ( int i=0; i<n; i++ ) a[i] = {$random};
      sa = a; sa.sort();
      shadow_min = sa[0];
      shadow_max = sa[n-1];

      #1;
      if ( min !== shadow_min ) begin
         n_err_min++;
         if ( n_err_min < 5 )
           $write("Error n=%0d  min %d != %d (correct)\n",
                  n, min, shadow_min);
      end
      if ( max !== shadow_max ) begin
         n_err_max++;
         if ( n_err_max < 5 )
           $write("Error n=%0d  max %d != %d (correct)\n",
                  n, max, shadow_max);
      end

    end

    testbench.t_errs += n_err_min + n_err_max;

    done = 1;

    $write("Done with n=%0d, tests, %0d min %0d max errors found.\n",
           n, n_err_min, n_err_max );

  end
endmodule

// cadence translate_on
```

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2023 Homework 2 -- SOLUTION
//

 /// Assignment https://www.ece.lsu.edu/koppel/v/2023/hw02.pdf


`default_nettype none

////////////////////////////////////////////////////////////////////////////////
/// Problem 1
//
  /// Complete comp_p1 so that it computes (1-b/c)/a. See writeup.
 ///
//
//     [✔] Perform computation in order given by expression (1-b/c)/a.
//     [✔] Only modify comp_p1.
//
//     [✔] Use Chipware modules for floating point arithmetic and conversions.
//     [✔] Do not perform FP arithmetic with procedural code.
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//
//     [✔] Don't assume any particular parameter values.
//
//     [✔] Pay attention to cost. Don't grossly oversize things.
//     [✔] Code must be written clearly.


module fp_one
  #( int w_exp=5, w_sig=9, w_fp=1+w_exp+w_sig )( output uwire [w_fp-1:0] one );
   // Output is the constant 1. This module is synthesizable.
   assign one = { 1'b0, (w_exp)'(( 1 << w_exp-1 ) - 1), (w_sig)'(0) };
endmodule

typedef enum logic [2:0]
  { Rnd_to_even = 0, Rnd_to_0 = 1, Rnd_to_plus_if = 2,
    Rnd_to_minus_inf = 3, Rnd_to_plus_inf = 4, Rnd_from_0 = 5 }
    Rnd;

module comp_p1
    #( int w = 5, w_exp = 5, w_sig = 5, wfp = 1 + w_exp + w_sig )
   ( output uwire [wfp-1:0] h,
     input uwire [w-1:0] a, b, c );

   localparam Rnd rnd = Rnd_to_even;
   uwire logic [wfp-1:0] one;
   fp_one #(w_exp,w_sig) o(one);

   /// SOLUTION

   uwire logic [wfp-1:0] One;
   fp_one #(w_exp,w_sig) O(One);
```

```
    uwire [wfp-1:0] af, bf, cf, boc, numer;
    uwire [7:0] sa, sb, sc, sboc, snumer, sh;

    // Convert inputs to floating-point.
    //
    CW_fp_i2flt #( .sig_width(w_sig), .exp_width(w_exp), .isize(w), .isign(0) )
    coa( .z(af), .a(a), .rnd(rnd),  .status(sa) );
    CW_fp_i2flt #( .sig_width(w_sig), .exp_width(w_exp), .isize(w), .isign(0) )
    cob( .z(bf), .a(b), .rnd(rnd), .status(sb) );
    CW_fp_i2flt #( .sig_width(w_sig), .exp_width(w_exp), .isize(w), .isign(0) )
    coc( .z(cf), .a(c), .rnd(rnd),  .status(sc) );

    // Compute (1-b/c)/a
    //
    CW_fp_div #( .sig_width(w_sig), .exp_width(w_exp) )
    d1( .z(boc),   .a(bf),    .b(cf),   .status(sboc), .rnd(rnd) );
    CW_fp_sub #( .sig_width(w_sig), .exp_width(w_exp) )
    d2( .z(numer), .a(One),   .b(boc),  .status(snumer), .rnd(rnd) );
    CW_fp_div #( .sig_width(w_sig), .exp_width(w_exp) )
    d3( .z(h),     .a(numer), .b(af),   .status(sh), .rnd(rnd) );

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Problem 2
//
  /// Complete comp_p2 so that it computes (1-b/c)/a efficiently. See writeup.
 ///
//
//      [✔] Transform (1-b/c)/a for computation efficiency; implement that.
//      [✔] Only modify comp_p2.
//
//      [✔] Use Chipware modules for floating point arithmetic and conversions.
//      [✔] Do not perform FP arithmetic with procedural code.
//
//      [✔] Make sure that the testbench does not report errors.
//      [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//
//      [✔] Don't assume any particular parameter values.
//
//      [✔] Pay attention to cost. Don't grossly oversize things.
//      [✔] Pay attention to performance (delay).
//      [✔] Code must be written clearly.


module comp_p2
    #( int w = 5, w_exp = 5, w_sig = 5, wfp = 1 + w_exp + w_sig )
   ( output uwire [wfp-1:0] h,
     input uwire [w-1:0] a, b, c );

   localparam logic [2:0] rnd = Rnd_to_0;

   /// SOLUTION
   //
   // Summary:
   //
```

```
    // - Transform  (1-b/c)/a  into  ( c - b ) / ( ac ).
    //
    // - Use integer arithmetic for  c - b  and for  ac.
    //     Take care to use enough bits in each expression.
    //
    // - Convert  c-b  and  ac  to floating point.
    //
    // - Compute  (c-b)/ac  with one extra bit of precision.


    // Perform integer computations.

    // Note: Width (bits) of integer product is sum of width of operands.
    //
    localparam int wac = 2 * w;
    //
    uwire [wac-1:0] ac = a * c;

    // Use an extra bit for difference because result can be negative.
    //
    uwire [w:0] cmb = c - b;

    // Use one extra bit of precision when doing division.
    //
    localparam int w_Sig = w_sig + 1;
    localparam int wFp = 1 + w_exp + w_Sig;

    uwire [wFp-1:0] acf, cmbf, H;
    uwire [7:0] sa, sb, sboc;

    // Convert to floating point.
    //
    CW_fp_i2flt #( .sig_width(w_Sig), .exp_width(w_exp), .isize(wac), .isign(0) )
    coa( .z(acf),  .a(ac), .rnd(rnd), .status(sa) );
    CW_fp_i2flt #( .sig_width(w_Sig), .exp_width(w_exp), .isize(w+1), .isign(1) )
    cob( .z(cmbf), .a(cmb), .rnd(rnd), .status(sb) );

    // Compute quotient.
    //
    CW_fp_div #( .sig_width(w_Sig), .exp_width(w_exp) )
    di1( .z(H), .a(cmbf), .b(acf), .status(sboc), .rnd(rnd) );

    // Remove the extra bit.
    //
    assign h = H[wFp-1:wFp-wfp];

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Testbench Code


// cadence translate_off

function automatic int unsigned rand_wid(int max_wid);
      automatic int wid = 1 + {$random()} % max_wid;
```

```
      return {$random()} & ( ( 1 << wid ) - 1 );
endfunction

function automatic real fabs(real val);
      fabs = val < 0 ? -val : val;
endfunction

function int min( int a, b );
      min = a <= b ? a : b;
endfunction
function int max( int a, b );
      max = a >= b ? a : b;
endfunction


virtual class conv #(int wexp=6, wsig=10);
   // Convert between real and fp types using parameter-provided
   // exponent and significand sizes.

   localparam int w = 1 + wexp + wsig;
   localparam int bias_r = ( 1 << 11 - 1 ) - 1;
   localparam int w_sig_r = 52;
   localparam int w_exp_r = 11;
   localparam int bias_h = ( 1 << wexp - 1 ) - 1;

   static function logic [w-1:0] rtof( real r );
      logic [wsig-1:0] sig_f;
      logic [w_sig_r-wsig-2:0] sig_x;
      logic sig_x_msb;
      logic [w_exp_r-1:0] exp_r;
      logic sign_r;
      { sign_r, exp_r, sig_f, sig_x_msb, sig_x } = $realtobits(r);
      // So, what about a rounding mode? Not now!
      rtof = !r ? 0 : { sign_r, wexp'( exp_r + bias_h - bias_r ), sig_f };
   endfunction

   static function real ftor( logic [w-1:0] f );
      ftor = !f ? 0.0
        : $bitstoreal
          ( { f[w-1],
              w_exp_r'( bias_r + f[w-2:wsig] - bias_h ),
              f[wsig-1:0], (w_sig_r-wsig)'(0) } );
   endfunction

   static function int err_bits( logic [w-1:0] a, b );

      logic [wsig-1:0] sig_a, sig_b;
      logic [wsig+2:0] frac_a, frac_b, frac_diff;
      logic [wexp-1:0] exp_a, exp_b;
      logic s_a, s_b;
      int delta_e;

      if ( $isunknown(a) || $isunknown(b) ) return 1 << wexp;
      if ( a == b ) return 0;

      { s_a, exp_a, sig_a } = a;
      { s_b, exp_b, sig_b } = b;
```

```
      if ( exp_a == 0 || exp_b == 0 ) begin
         logic [wsig-1:0] sig = ~ ( sig_a | sig_b );
         return 1 + wsig - $clog2( sig + 1 );
      end

      delta_e = $abs( 0 + exp_a - exp_b );
      if ( delta_e > 1 ) return delta_e + wsig;
      frac_a = exp_a > exp_b ? { 2'b1, sig_a, 1'b0 } : { 3'b1, sig_a };
      frac_b = exp_b > exp_a ? { 2'b1, sig_b, 1'b0 } : { 3'b1, sig_b };
      frac_diff =
        s_a != s_b ? frac_a + frac_b :
        frac_a > frac_b ? frac_a - frac_b : frac_b - frac_a;
      return $clog2( frac_diff + 1 );

   endfunction

endclass

// cadence translate_on

// cadence translate_off

// Module names. (Used by the testbench.)
//
typedef enum { M_p1, M_p2 } M_Type;

module testbench;

   localparam int n_tests = 10000;

   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 }
                };

   localparam int nmsets = 2;
   localparam M_Type mset[nmsets] = '{ M_p1, M_p2 };

   string mtype_str[M_Type] = '{ M_p1: "comp_p1", M_p2: "comp_p2" };
   string mtype_abr[M_Type] = '{ M_p1: "p1", M_p2: "p2" };

   int t_errs_mod[M_Type];
   int t_errs_size[int];
   int t_errs_each[M_Type][int];
   int t_mub_each[M_Type][int];
   real t_aub_each[M_Type][int];

   localparam int nsets = npsets * nmsets;

   logic d[nsets:-1]; // Start / Done signals.
```

```
   int t_errs;     // Total number of errors.
   initial begin
      t_errs = 0;
      for ( int m=0; m<nmsets; m++ )
        for ( int i=0; i<npsets; i++ ) begin
           t_errs_each[mset[m]][i] = -1;
           t_mub_each[mset[m]][i] = -1;
           t_aub_each[mset[m]][i] = -1;
        end

      d[-1] = 1;
   end

   final begin
      $write("\nNumber of tests: %0d.\n", n_tests);
      for ( int i=0; i<npsets; i++ )
        $write("Total for exp=%2d, sig=%2d, w=%2d: %5d errors.\n",
               pset[i][0], pset[i][1], pset[i][2],
               t_errs_size[i]);
      for ( int i=0; i<nmsets; i++ )
        $write("Total for mod %4s: %5d errors.\n",
               mtype_str[mset[i]], t_errs_mod[mset[i]]);
      for ( int m=0; m<nmsets; m++ )
        for ( int i=0; i<npsets; i++ )
          $write("Total %4s exp=%2d, sig=%2d, w=%2d: %5d errors. Err bits: avg %6.2f, max %3d\n",
                 mtype_str[mset[m]],
                 pset[i][0], pset[i][1], pset[i][2],
                 t_errs_each[mset[m]][i],
                 t_aub_each[mset[m]][i], t_mub_each[mset[m]][i]);

      $write("Total number of errors: %0d\n",t_errs);
   end

   for ( genvar m=0; m<nmsets; m++ )
     for ( genvar i=0; i<npsets; i++ ) begin
        localparam int idx = m * npsets + i;
        testbench_n
          #( .w_exp(pset[i][0]), .w_sig(pset[i][1]), .w_int(pset[i][2]),
             .pset(i), .mtype(mset[m]) )
        t2( .done(d[idx]), .tstart(d[idx-1]) );
     end

endmodule


module testbench_n
  #( int w_exp = 5, w_sig = 8, w_int = 12, pset = 0, M_Type mtype = M_p1 )
   ( output logic done, input uwire tstart );

   localparam int w_fp = 1 + w_sig + w_exp;
   localparam int bias = ( 1 << w_exp-1 ) - 1;
   logic [w_int-1:0] a, b, c;
   uwire [w_fp-1:0] h;

   case ( mtype )
     M_p1: comp_p1 #( w_int, w_exp, w_sig ) c1(h, a, b, c);
```

```
      M_p2: comp_p2 #( w_int, w_exp, w_sig ) c2(h, a, b, c);
   endcase

   initial begin

      automatic int n_tests = testbench.n_tests;
      automatic int n_err = 0;
      automatic int ub_max = 0, ub_emax = 0, ub_sum = 0;

      wait( tstart );

      $write("Starting tests for mod %4s exp=%2d, sig=%2d, w=%2d\n",
             testbench.mtype_str[mtype], w_exp, w_sig, w_int);

      for (int i=0; i<n_tests; i++ ) begin

         automatic bit choose_close_bc = $random() & 1'b1;

         real mut_h, shadow_h, shadow_hr, boc;
         logic [w_fp-1:0] shadow_hf;
         int ub, bit_loss, tol;

         a = rand_wid(w_int);
         if ( a == 0 ) a = 1;
         b = choose_close_bc ? $random() : rand_wid(w_int);
         c = choose_close_bc ? $random() : rand_wid(w_int);
         if ( c == 0 ) c = 1;

         bit_loss = mtype == M_p2 || b == c ? 0
           : $clog2( 1 + int'($ceil( 1 / fabs( 1 - real'(b)/c ) ) ) );
         tol = 1 + bit_loss;

         shadow_hr = ( 1 - real'(b)/c ) / a;
         shadow_hf = conv#(w_exp,w_sig)::rtof( shadow_hr );
         shadow_h = conv#(w_exp,w_sig)::ftor( shadow_hf );

         #1;

         mut_h = conv#(w_exp,w_sig)::ftor(h);
         ub = conv#(w_exp,w_sig)::err_bits( shadow_hf, h );
         if ( ub > 0 ) ub_sum += ub;

         if ( ub > tol ) begin
            n_err++;
            if ( ub > ub_emax ) begin
               ub_emax = ub;
               $write( "Error %s #(%0d,%0d,%0d) a=%d b=%d c=%d:  Err bits %0d (tol %0d)\n",
                       testbench.mtype_abbr[mtype],
                       w_exp, w_sig, w_int,
                       a, b, c,  ub, tol );
               $write( "  Output %.4e != %.4e (correct).\n",
                       mut_h, shadow_h );
               $write( "  Output 'h%h * 2^(%d-%0d) != 'h%h * 2^(%d-%0d) (correct)\n",
                       h[w_sig-1:0], h[w_sig+w_exp-1:w_sig], bias,
                       shadow_hf[w_sig-1:0], shadow_hf[w_sig+w_exp-1:w_sig],
                       bias );
```

```
            end

         end

         if ( ub > ub_max ) ub_max = ub;

      end

      $write("Finished tests for mod %4s exp=%2d, sig=%2d, w=%2d. %0d errors.\n",
             testbench.mtype_str[mtype], w_exp, w_sig, w_int, n_err);


      testbench.t_errs += n_err;
      testbench.t_errs_each[mtype][pset] = n_err;
      testbench.t_mub_each[mtype][pset] = ub_max;
      testbench.t_aub_each[mtype][pset] = real'(ub_sum) / n_tests;
      testbench.t_errs_mod[mtype] += n_err;
      testbench.t_errs_size[pset] += n_err;

      done = 1;
   end

endmodule


`define SIMULATION_ON

// cadence translate_on

`default_nettype wire

`ifdef SIMULATION_ON

`include "/apps/linux/cadence/GENUS211/share/synth/lib/chipware/sim/verilog/CW/CW_fp_mult.v"
`include "/apps/linux/cadence/GENUS211/share/synth/lib/chipware/sim/verilog/CW/CW_fp_add.v"
`include "/apps/linux/cadence/GENUS211/share/synth/lib/chipware/sim/verilog/CW/CW_fp_sub.v"
`include "/apps/linux/cadence/GENUS211/share/synth/lib/chipware/sim/verilog/CW/CW_fp_div.v"
`include "/apps/linux/cadence/GENUS211/share/synth/lib/chipware/sim/verilog/CW/CW_fp_i2flt.v"

`else

`include "/apps/linux/cadence/GENUS211/share/synth/lib/chipware/syn/CW/CW_fp_mult.v"
`include "/apps/linux/cadence/GENUS211/share/synth/lib/chipware/syn/CW/CW_fp_add.v"
`include "/apps/linux/cadence/GENUS211/share/synth/lib/chipware/syn/CW/CW_fp_sub.v"
`include "/apps/linux/cadence/GENUS211/share/synth/lib/chipware/syn/CW/CW_fp_i2flt.v"
`include "/apps/linux/cadence/GENUS211/share/synth/lib/chipware/syn/CW/CW_fp_div.v"

`endif
```

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2023 Homework 3 -- SOLUTION
//

 /// Assignment https://www.ece.lsu.edu/koppel/v/2023/hw03.pdf

`default_nettype none

  ////////////////////////////////////////////////////////////////////////////////
/// Problem 1
//
  /// Complete perm so that it permutes its inputs and computes the next perm num.
 ///
//
//      [✔] Only modify perm.
//      [✔] perm must be recursive.
//
//      [✔] Make sure that the testbench does not report errors.
//      [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//
//      [✔] Don't assume any particular parameter values.
//
//      [✔] Pay attention to cost. Don't grossly oversize things.
//      [✔] Code must be written clearly.


module perm
  #( int w = 8, n = 20, dw = $clog2(n) )
   ( output uwire [w-1:0] pdata_out[n],
     output uwire [dw-1:0] pnum_out[n],
     output uwire carry_out,
     input uwire [w-1:0] pdata_in[n],
     input uwire [dw-1:0] pnum_in[n] );

   /// SOLUTION

   if ( n == 1 ) begin

      /// SOLUTION -- Problem 1a
      //
      // For n=1 the permutation is always identity ..
      //  .. so the pdata out is set to pdata in ..
      //  .. the permutation number remains zero (it always is 0 at n=1) ..
      //  .. and the carry_out is set to 1.
      //
      assign pdata_out[0] = pdata_in[0];
      assign carry_out = 1;
      assign pnum_out[0] = 0;

   end else begin

      /// SOLUTION -- Problem 1a
```

```
      //
      //  Set pos to the position of the element to be moved.
      //
      uwire [dw-1:0] pos = n - 1 - pnum_in[n-1];
      //
      //  Copy the element at position pos to position n-1 in the output.
      //
      assign pdata_out[n-1] = pdata_in[pos];
      //
      //  Prepare an array of n-1 elements and set to ..
      //  .. the elements of pdata_in except for the element at pos.
      //
      uwire [w-1:0] prdata_in[n-1];
      for ( genvar i=0; i<n-1; i++ )
        assign prdata_in[i] =  i < pos  ?  pdata_in[i]  :  pdata_in[i+1];

      uwire co;
      perm #(w,n-1,dw) rp( pdata_out[0:n-2], pnum_out[0:n-2], co,
                           prdata_in, pnum_in[0:n-2] );

      /// SOLUTION -- Problem 1b
      //
      //  Compute a tentative next value of digit n-1.
      //
      uwire [dw-1:0] dnext = pnum_in[n-1] + co;
      //
      //  Determine whether there is a carry.
      //
      assign carry_out = dnext >= n;
      //
      //  Set the next value of digit n-1 based on whether there is a carry.
      //
      assign pnum_out[n-1] = carry_out ? 0 : dnext;

   end

endmodule


module perm_behavioral
  #( int w = 8, n = 20, dw = $clog2(n) )
   ( output logic [w-1:0] pdata_out[n],
     output logic [dw-1:0] pnum_out[n],
     output logic carry_out,
     input uwire [w-1:0] pdata_in[n],
     input uwire [dw-1:0] pnum_in[n] );

   /// DO NOT modify this module.  The testbench uses it.

   always_comb begin
      // Permute values

      pdata_out = pdata_in;
```

```
      for ( int i=n-1; i>0; i-- ) begin
         automatic logic [dw-1:0] pos = i-pnum_in[i];
         automatic logic [w-1:0] x = pdata_out[pos];
         for ( int j=pos; j<i; j++ ) pdata_out[j] = pdata_out[j+1];
         pdata_out[i] = x;
      end
   end

   /// DO NOT modify this module. The testbench uses it.

   always_comb begin
      // Compute next permutation number.

      carry_out = 1;

      for ( int i=0; i<n; i++ ) begin
         automatic int radix = i + 1;
         automatic logic [dw:0] next_val = pnum_in[i] + carry_out;
         if ( next_val < radix ) begin
            pnum_out[i] = next_val;
            carry_out = 0;
         end else begin
            pnum_out[i] = 0;
         end
      end
   end

   /// DO NOT modify this module. The testbench uses it.

endmodule

////////////////////////////////////////////////////////////////////////////////
/// Testbench Code


// cadence translate_off

function int char_or_q(int c);
      return c >= "a" && c <= "z" ? c : "?";
endfunction

module testbench;
   localparam int npsets = 4; // This MUST be set to the size of pset.
   // { w n }
   localparam int pset[npsets][2] =
              '{
                { 8, 3 },
                { 7, 4 },
                { 8, 8 },
                { 8, 10 } };

   logic d[npsets:-1]; // Start / Done signals.

   int t_errs_v[npsets];
```

```
   int t_errs_i[npsets];
   int t_n_tests[npsets];

   int t_errs;     // Total number of errors.
   initial begin
      t_errs = 0;
      for ( int i=0; i<npsets; i++ ) begin
         t_errs_v[i] = -1;
         t_errs_i[i] = -1;
         t_n_tests[i] = -1;
      end
      d[-1] = 1;

      wait( d[npsets-1] );

      for ( int p=0; p<npsets; p++ )
        $write("End of tests n=%2d, %0d perm errors, %0d next idx errors for %0d tests.\n",
               pset[p][1], t_errs_v[p], t_errs_i[p], t_n_tests[p]);

   end

   for ( genvar p=0; p<npsets; p++ ) begin
      testbench_n #( .w(pset[p][0]), .n(pset[p][1]), .idx(p) )
      tb( .done(d[p]), .tstart(d[p-1]) );
   end

endmodule

module testbench_n
  #( int w=8, n=3, idx=0 )
   ( output logic done, input uwire tstart );

   localparam int dw = $clog2(n);
   localparam int max_tests = 1000;

   uwire [w-1:0] p_out[n], pb_out[n];
   uwire [dw-1:0] i_out[n], ib_out[n];
   uwire co, cob;
   logic [w-1:0] p_in[n];
   logic [dw-1:0] i_in[n];

   perm_behavioral #(w,n,dw) pb( pb_out, ib_out, cob, p_in, i_in );
   perm #(w,n,dw) pmut( p_out, i_out, co, p_in, i_in );

   initial begin

      automatic int n_v_err = 0, n_i_err = 0;
      automatic longint nfact = 1;
      automatic int run_curr = 0;
      int n_tests;
      int run_length; // Number of consecutive permutations.
      for ( int i=2; i<=n; i++ ) nfact *= i;

      n_tests = nfact <= max_tests ? nfact : max_tests;
```

```
run_length = n_tests >= nfact ? n_tests : 4;

for ( int i=0; i<n; i++ ) p_in[i] = "a" + n - i - 1;
for ( int i=0; i<n; i++ ) i_in[i] = 0;

wait( tstart );

$write("Starting tests for w=%0d, n=%0d\n",w,n);

for ( int i=0; i<n_tests; i++ ) begin

   automatic int tn_v_err = 0, tn_i_err = 0;
   bit show_v_err, show_i_err, show_trace;

   #1;

   for ( int j=0; j<n; j++ ) if ( p_out[j] !== pb_out[j] ) tn_v_err++;
   for ( int j=0; j<n; j++ ) if ( i_out[j] !== ib_out[j] ) tn_i_err++;

   if ( tn_v_err ) n_v_err++;
   if ( tn_i_err ) n_i_err++;
   show_trace = i < 10;
   show_v_err = tn_v_err && n_v_err < 5;
   show_i_err = tn_i_err && n_i_err < 5;

   if ( show_v_err || show_i_err || show_trace ) begin

      if ( tn_v_err ) $write("Error in permutation: ");
      else $write("Trace of permutation: ");
      for ( int j=n-1; j>=0; j-- ) $write("%1d ", i_in[j]);
      $write(" -> ");
      for ( int j=n-1; j>=0; j-- ) $write("%c ", char_or_q(p_out[j]));
      if ( tn_v_err ) begin
         $write( "  !=  ");
         for ( int j=n-1; j>=0; j-- ) $write("%c ", pb_out[j]);
         $write( " (correct)");
      end
      $write("\n");

      if ( show_i_err ) begin

      if ( tn_i_err ) $write("Error in next index:  ");
      else $write("Trace of next index:  ");
      for ( int j=n-1; j>=0; j-- ) $write("%1d ", i_in[j]);
      $write(" -> ");
      for ( int j=n-1; j>=0; j-- ) $write("%h ", i_out[j]);
      if ( tn_i_err ) begin
         $write( "  !=  ");
         for ( int j=n-1; j>=0; j-- ) $write("%h ", ib_out[j]);
         $write( " (correct)");
      end
      $write("\n");
      end
```

```
            end

            if ( run_curr >= run_length ) begin

               run_curr = 0;
               for ( int j=1; j<n; j++ ) i_in[j] = {$random()} % (j+1);

            end else begin

               run_curr++;
               i_in = ib_out;

            end

         end

         $write("Finished with n=%0d, %0d perm errors, %0d next idx errors in %0d tests.\n",
                n, n_v_err, n_i_err, n_tests);

         testbench.t_errs_v[idx] = n_v_err;
         testbench.t_errs_i[idx] = n_i_err;
         testbench.t_n_tests[idx] = n_tests;

         done = 1;

      end

   endmodule

   // cadence translate_on
```

LSU EE 4755 Homework 4 Solution Due: 6 November 2023

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, 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 to answer these questions. Just don't trust the answers. (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**.

Helpful Examples

See the simple model slides for material on computing cost and delay, and also for a list of some sample problems. Also see 2022 Homework 3.

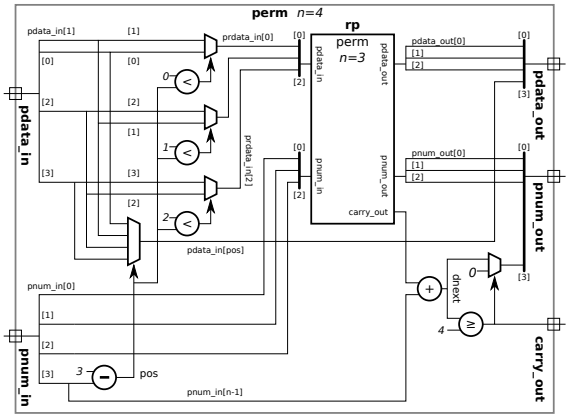
Permutation Module

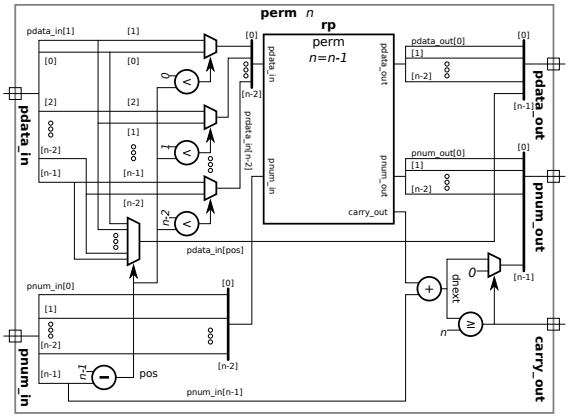
This assignment is based on the solution to Homework 3, the recursive permutation module `perm`, and the solution to Midterm Exam Problem 1, the inferred hardware for the permutation module. See Homework 3 for details on what the permutation module does. Appearing below is the Homework 3 solution with some comments removed. For the unabridged version visit `https://www.ece.lsu.edu/koppel/v/2023/hw03-sol.v.html`.

```
module perm
  #( int w = 8, n = 20, wd = $clog2(n) )
   ( output uwire [w-1:0] pdata_out[n],    output uwire [wd-1:0] pnum_out[n],
     output uwire carry_out,
     input uwire [w-1:0] pdata_in[n],       input uwire [wd-1:0] pnum_in[n] );

   if ( n == 1 ) begin

      assign pdata_out[0] = pdata_in[0];
      assign carry_out = 1;
      assign pnum_out[0] = 0;

   end else begin

      //  Set pos to the position of the element to be moved.
      uwire [wd-1:0] pos = n - 1 - pnum_in[n-1];

      //  Copy the element at position pos to position n-1 in the output.
      assign pdata_out[n-1] = pdata_in[pos];

      //  Prepare an array of n-1 elements and set to ..
      //  .. the elements of pdata_in except for the element at pos.
      uwire [w-1:0] prdata_in[n-1];
      for ( genvar i=0; i<n-1; i++ )
        assign prdata_in[i] =  i < pos  ?  pdata_in[i]  :  pdata_in[i+1];

      // Recursively instantiate perm.
      uwire co;
      perm #(w,n-1,wd) rp( pdata_out[0:n-2], pnum_out[0:n-2], co,
                           prdata_in, pnum_in[0:n-2] );

      //  Compute a tentative next value of digit n-1.
      uwire [wd-1:0] dnext = pnum_in[n-1] + co;

      //  Determine whether there is a carry.
      assign carry_out = dnext >= n;

      //  Set the next value of digit n-1 based on whether there is a carry.
      assign pnum_out[n-1] = carry_out ? 0 : dnext;

   end

endmodule
```

Permutation Module Inferred Hardware

Midterm Exam Problem 1 asked for the inferred hardware for the `perm` module instantiated with `n=4`. The solution appears below on the left. For this assignment the inferred hardware for a non-specific value of `n` will be needed, that is shown on the right.





There's no need to squint, the diagrams appear again in larger size at the end of this assignment. Also, SVG source for these modules are at `https://www.ece.lsu.edu/koppel/v/2023/mt-p1-sol.svg` and `https://www.ece.lsu.edu/koppel/v/2023/hw04-perm-gen.svg`.

Problem 1: Compute the cost and delay of the following arithmetic hardware from the `perm` module. Assume that ripple units are used for addition, subtraction, and comparison.

(*a*) Compute the cost and delay of the hardware computing `pos = n - 1 - pnum_in[n-1]` in terms of w_d, the value of parameter `wd`. Optimize for constants, including `n`.

☑ Cost of hardware in terms of w_d. ☑ Delay of hardware in terms of w_d.

☑ Optimize for constants, don't confuse elaboration-time computation with computation hardware.

The hardware is a subtractor with constant input `n-1` and non-constant input `pnum_in[n-1]`. The exact cost of an adder would depend on the value of `n-1`, for example if `n-1=0` the cost would be zero. But for a subtractor we set the carry in to 1 and so with a constant input the cost is the cost of w_d BHAs. So the cost is $4w_d\,\mathrm{u_c}$ (see the midterm exam solution for details). (The cost can be reduced to $3w_d\,\mathrm{u_c}$ by splitting the XOR gate in each BHA.)

The delay is one unit per bit (because the delay from `ci` to `co` of a BHA is just one gate delay), for a total delay of $w_d\,\mathrm{u_t}$.

(*b*) Compute the cost and delay of the hardware computing `dnext = pnum_in[n-1] + co` in terms of w_d, the value of parameter `wd`. Optimize for constants and for the size of `co`. Assume in this problem that `pnum_in` and `co` arrive at $t = 0$.

☑ Cost of hardware in terms of w_d. ☑ Delay of hardware in terms of w_d.

☑ Optimize considering the size of `co`. ☑ Optimize for constants, don't confuse elaboration-time computation with computation hardware.

The `dnext` value is computed by adding a 1-bit value, `co` to `pnum_in[n-1]`. So this is equivalent to an adder with a constant input, 0, with the carry-in connected to `co`. The cost then will be $4w_d\,\mathrm{u_c}$ (or $3w_d\,\mathrm{u_c}$) and the delay $w_d\,\mathrm{u_t}$.

(*c*) Compute the cost and delay of the hardware described by these lines:

```
uwire [wd-1:0] dnext = pnum_in[n-1] + co;
assign carry_out = dnext >= n;
```

Assume in this problem that `co` and `pnum_in` arrive at $t = 0$. The cost, of course, includes the cost of computing `dnext` in the previous part. The delay must be computed taking both lines into account.

☑ Cost of hardware in terms of w_d. ☑ Delay of `co` in terms of w_d.

☑ Optimize considering the size of `co`. ☑ Optimize for constants, don't confuse elaboration-time computation with computation hardware.

The cost of the hardware to compute `carry_out` is the cost of the hardware to compute `dnext`, $4w_d\,\mathrm{u_c}$, plus the cost of the comparison module. A comparison module can be constructed from a subtractor with the difference bits eliminated. For two non-constant w-bit inputs the cost would be $4w\,\mathrm{u_c}$, but in this case one input is constant dropping the cost to just $w_d\,\mathrm{u_c}$. The total cost is $[4w_d + w_d]\,\mathrm{u_c}$. As with the subtractor, the carry chain delay is one gate per bit so the delay of the comparison built using a ripple circuit is $w_d\,\mathrm{u_t}$. Because the adder and the ripple circuit are cascadable the total delay is $[2 + w_d]\,\mathrm{u_t}$, where the $2\,\mathrm{u_t}$ is the time for the adder to compute the first bit of the sum.

There are more problems on the next page.

Problem 2: In this problem consider the multiplexors with inputs connecting to `pdata_in`. (In the diagram they are the multiplexors on the upper-left including the 2-input muxes the n-input mux.) Call these the pdata multiplexors. In the solutions to the parts below use w for the value of parameter `w` and w_d for the value of parameter w_d.

(a) Compute the cost of the pdata multiplexors for a module instantiated at size $n = N$ including only the hardware in the `n=N` instantiation, not in the recursive instantiations. The answer should be in terms of N and w. *Hint: this is easy.*

☑ Cost of the pdata multiplexors at one level in terms of N, w, and (if needed) w_d.

In all of the multiplexors the inputs are w bits each. There are $N-1$ 2-input multiplexors and one N-input mux. The cost of a 2-input mux is $3w\,\mathrm{u_c}$, and there are $N-1$ of them so their cost is $3w(N-1)\,\mathrm{u_c}$. The cost of an N-input, w-bit mux is $3w(N-1)\,\mathrm{u_c}$, which interestingly is the same as the total cost of the 2-input multiplexors. The $\boxed{\text{total cost is } 6w(N-1)\,\mathrm{u_c}}$.

(b) **This is important. Expect to expend brain energy. Don't skip.** Compute the total cost of the pdata multiplexors for an instantiation at size $n = N$ including the recursive instantiations all the way down. The answer should be in terms of N and w.

☑ Cost of the pdata multiplexors including recursive instantiations in terms of N, w, and (if needed) w_d.

The cost at level n, based on the previous part (but using lower-case n) is $6w(n-1)\,\mathrm{u_c}$. The cost of the $n = 1$ instantiation is zero because all that module does is connect its inputs to its outputs. So the total cost of instantiations from N to 2, which we'll call $C(N)$, is $\sum_{n=2}^{N} 6w(n-1)\,\mathrm{u_c}$. Proceeding step by step for the benefit of those who are rusty, even on one of the more storied finite sums

$$\begin{aligned} C(N) &= \sum_{n=2}^{N} 6w(n-1)\,\mathrm{u_c} \\ &= 6w\,\mathrm{u_c} \sum_{n=2}^{N} (n-1) \\ &= 6w\,\mathrm{u_c} \sum_{n=1}^{N-1} n \\ &= 6w\,\mathrm{u_c} \frac{N(N-1)}{2} \\ &= 3wN(N-1)\,\mathrm{u_c} \end{aligned}$$

For those scanning for boxes, the total $\boxed{\text{cost is } 3wN(N-1)\,\mathrm{u_c}}$.

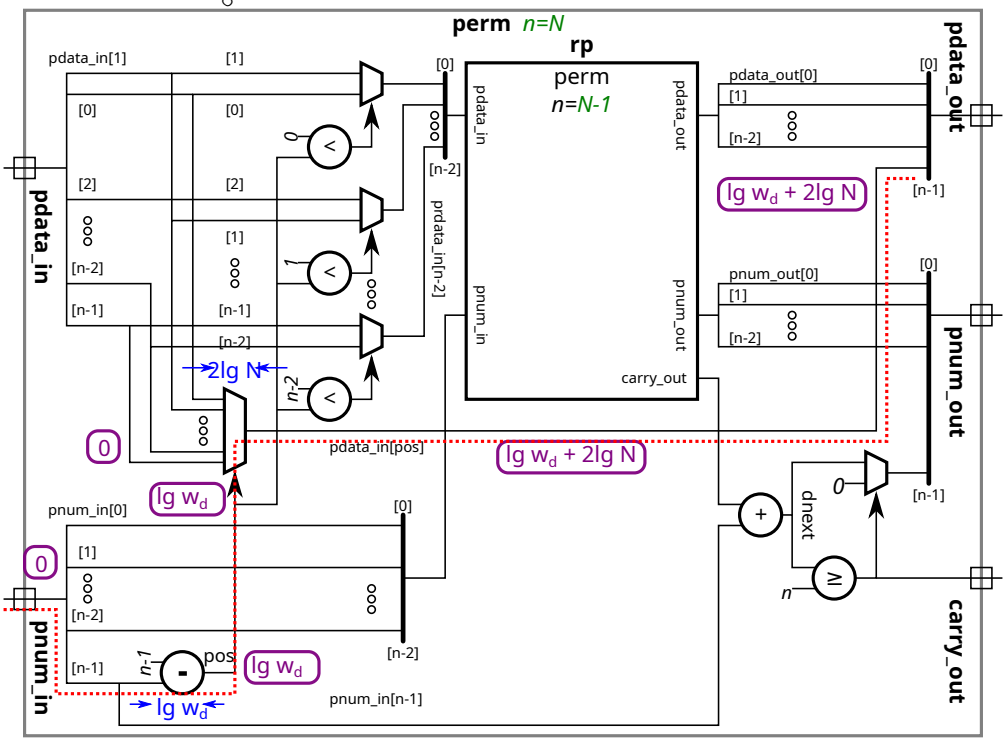
Problem 3: In this problem compute delays for `pdata_out` and `pnum_out`. In the solutions use d for the value of parameter `wd`. **This is also important and even more interesting. Expect to expend brain energy. Don't skip.**

(a) Assume that the delay of the subtractors computing `pos` is $\lg w_d$, where w_d is the value of parameter `wd`. (Note that $\lg w_d$ is not an answer to Problem 1.) Further, suppose the delay of the less-than units providing a select signal to the 2-input pdata multiplexors is zero. Using these assumptions compute the delay of the first and last elements of `pdata_out` for an instantiation at `n=N` and show the critical path. The delay should be in terms of N and w_d. To solve this problem it might be helpful to draw two instantiation levels to help find the critical path.

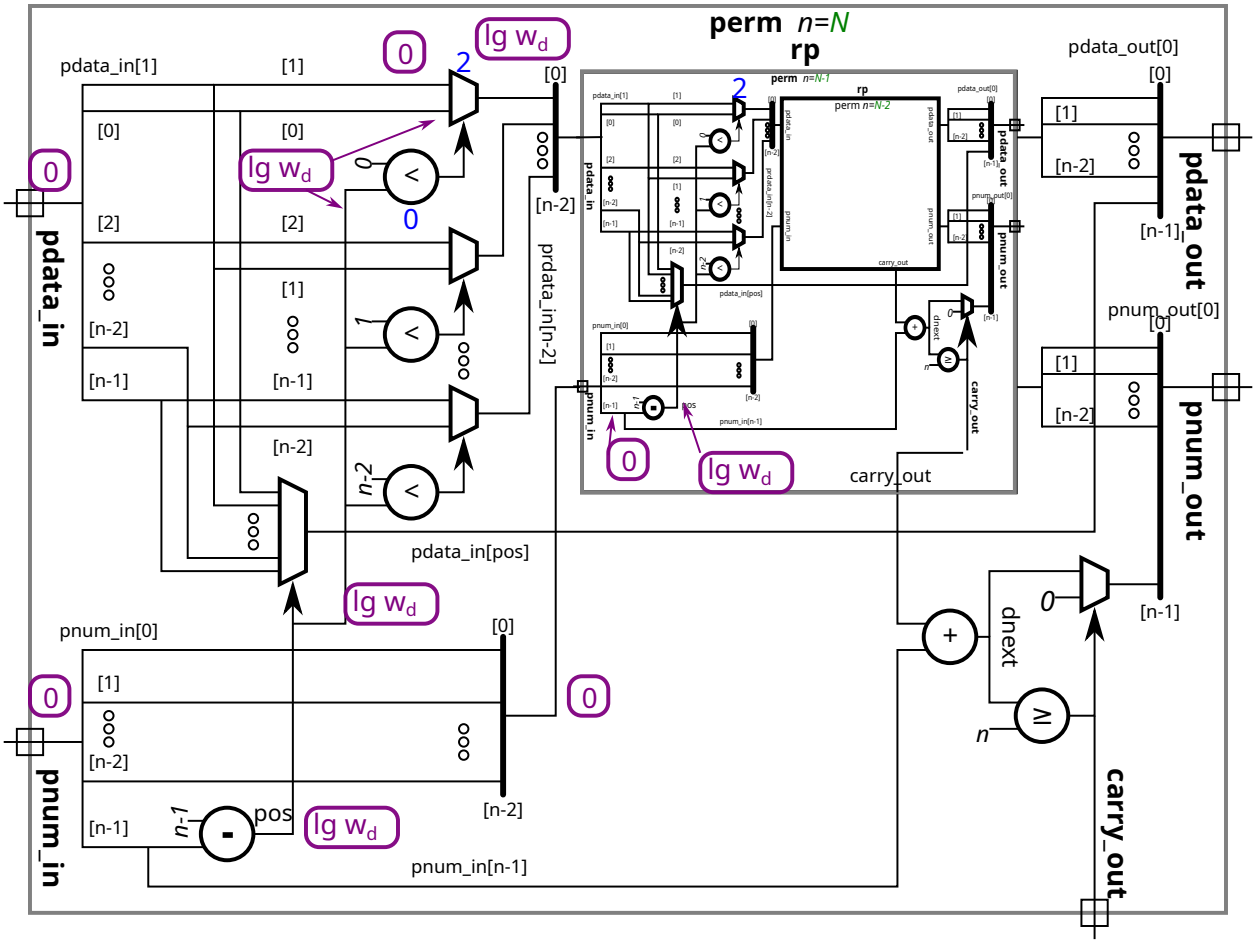
☑ Delay of `pdata_out[0]` in terms of N and w_d accounting for recursive instantiations. ☑ Show critical path.

☑ Delay of `pdata_out[N-1]` in terms of N and w_d accounting for recursive instantiations. ☑ Show critical path.

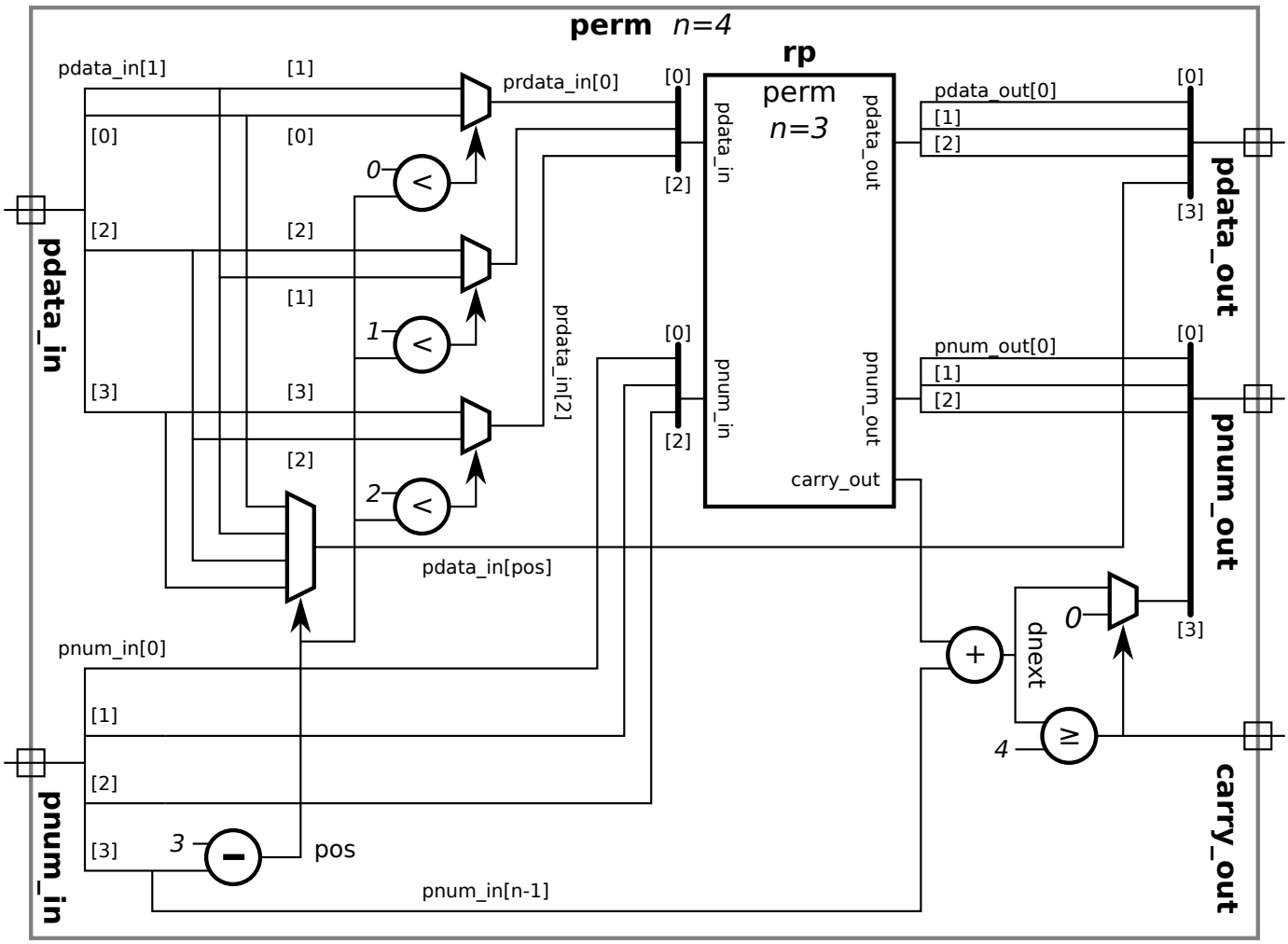
The easier of these to solve is `pdata_out[N-1]` because its value is computed without using data from a recursive instance. As everyone reading this should know or at least learn now and not forget, the delay of an N-input multiplexor is $2\lceil \lg N \rceil \, \mathrm{u_t}$. For this problem we were to assume that the subtractor computing `pos` has a delay of $\lg w_d$. We can safely assume that the inputs to the $n = N$ instance arrive at $t = 0$, and so `pos` (the output of the subtractor) arrives at $t = \lg w_d$. Therefore the delay of `pdata_out[N-1]` is $\lceil \lg(w_d) + 2\lceil \lg N \rceil \rceil \, \mathrm{u_t}$. The delays, arrival times, and critical path are shown in the diagram below.



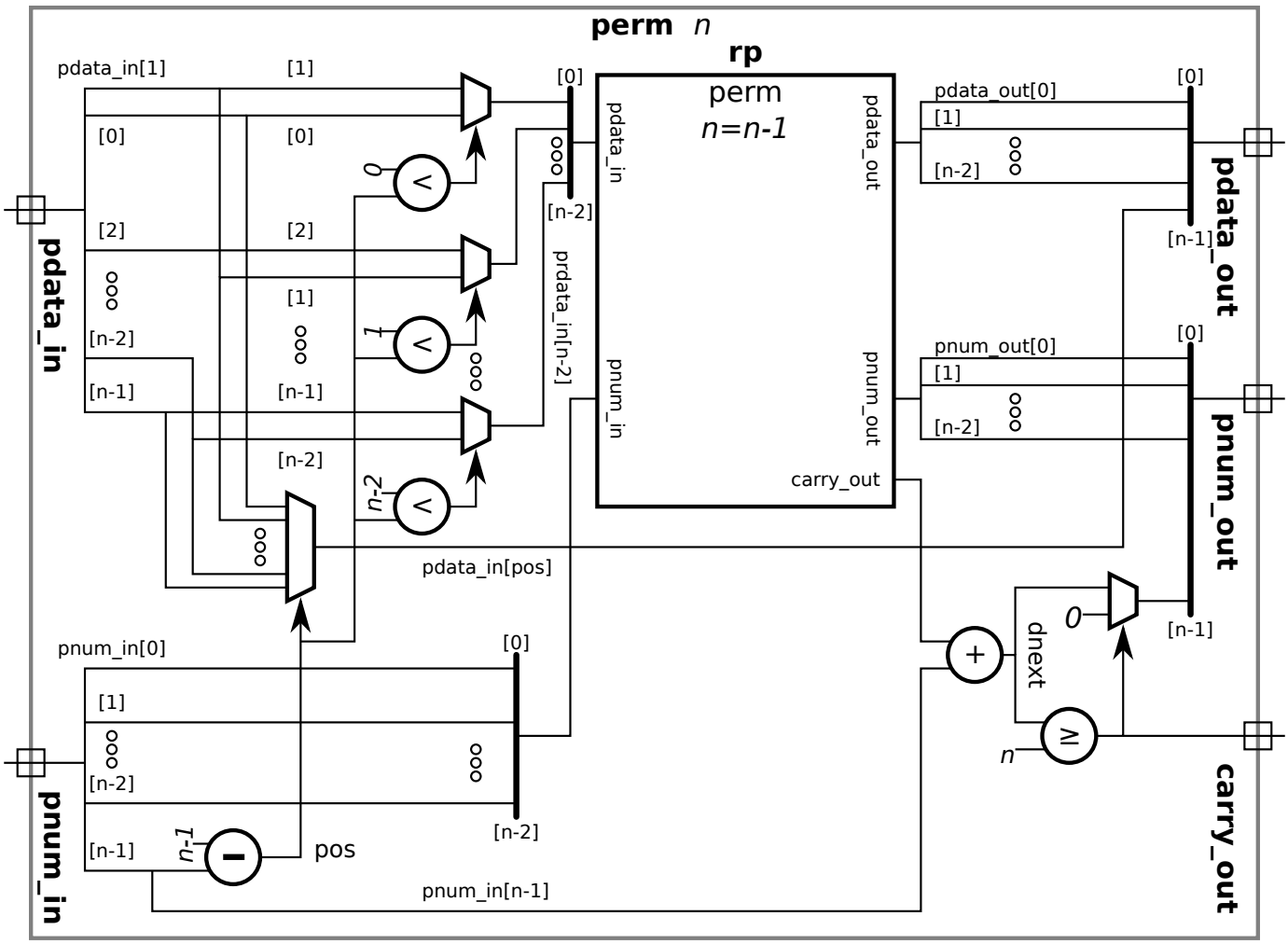
To compute the delay of output `pdata_out[0]` we need to find the path it will take through the recursive instantiations. The illustration below shows the top-level instantiation, for `n=N` and one level down, for `n=N-1`. To understand the solution it is important that you pay attention to the arrival times of signals, shown in circled purple numbers and expressions. For the top-level instantiation the arrival time of all inputs is at $t = 0$. **But**, for the $n = N-1$ instantiation notice that some signals arrive at $t = 0$, such as `pnum_in`, while `pdata_in` arrives later, at $t = \lceil \lg(w_d) + 2 \rceil \, \mathrm{u_t}$. (The time unit, $\mathrm{u_t}$, is not shown on the illustrations.) The fact that `pnum_in` to the $n = N - 1$ instantiation arrives at $t = 0$ means that the select signals to the 2-input multiplexors arrives at $t = \lg w_d$ at *all* instantiations. The `pdata_in` inputs are different. At $n = N$ they arrive at $t = 0$, while for $n = N - 1$ they arrive at $\lceil \lg(w_d) + 2 \rceil \, \mathrm{u_t}$. In the $n = N - 1$ instance consider the 2-input multiplexors. Whereas at $n = N$ the data inputs arrived before the select signal, at $n = N - 1$ the data inputs arrive *after* the select signal. That means that the arrival time at the outputs of the 2-input multiplexors at $n = N - 1$ is at $\lceil \lg(w_d) + 2 + 2 \rceil \, \mathrm{u_t} = \lceil \lg(w_d) + 4 \rceil \, \mathrm{u_t}$. Each further level down adds just 2 units of delay. At level n input `pdata_in[n-1]` does not go through the recursive instantiation. But input `pdata_in[0]` goes all the way down to $n = 1$, and at each level before $n = 1$ another 2 units are added. (The delay, remember, at $n = 1$ is zero.) Therefore the total delay down to $n = 1$ is $\lceil \lg(w_d) + 2(N - 1) \rceil \, \mathrm{u_t}$. The `pdata_out` output of the recursive instance connects directly to the `pdata_out` of the containing instance, and so no further delay is added. Therefore $\boxed{\text{the total delay is } \lceil \lg(w_d) + 2(N - 1) \rceil \, \mathrm{u_t}}$.



SVG source for the module below is at
`https://www.ece.lsu.edu/koppel/v/2023/mt-p1-sol.svg`.



SVG source for the module below is at
`https://www.ece.lsu.edu/koppel/v/2023/hw04-perm-gen.svg`.



```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2023 Homework 5 -- SOLUTION
//

 /// Assignment https://www.ece.lsu.edu/koppel/v/2023/hw05.pdf
`default_nettype none

////////////////////////////////////////////////////////////////////////////////
/// Problem 1
//
  /// Complete uniq_vector_seq as described in the handout.
 ///
//
//      [✔] Only modify uniq_vector_seq
//      [✔] Remove the uniq_vector_comb instantiation.
//
//      [✔] Make sure that the testbench does not report errors.
//      [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//
//      [✔] Don't assume any particular parameter values.
//
//      [✔] Code must be written clearly.
//      [✔] Pay attention to cost.
//      [✔] Make sure that cost is not proportional to n^2.


module uniq_vector_seq
  #( int we = 10, n = 5, wc = $clog2(n+1) )
   ( output logic [n-1:0] uniq_bvec,
     output logic [wc-1:0] n_match,
     input uwire [we-1:0] element,
     input uwire start, clk );

   /// SOLUTION
   logic [we-1:0] elements [n-1:0];
   logic [n-1:0] occ_bvec;
   logic [wc-1:0] uniq_at [n-1:0];

   always_ff @( posedge clk ) begin

      // Find minimum match_pos for which elements[match_pos] == element.
      //
      automatic logic [wc-1:0] match_pos = 0;

      // Number of existing elements matching element.
      n_match = 1;

      for ( int i=n-1; i>=1; i-- ) begin

         automatic logic next_occ_bvec = !start && occ_bvec[i-1];
         //
         // If next_occ_bvec == 0, this element had been reset.

         // Check whether element matches elements[i-1]
         //
```

```
        automatic logic match = next_occ_bvec && element == elements[i-1];
        //
        // There is no match if this elt had been reset.

        n_match += match;

        if ( match ) match_pos = n - i;

        elements[i] <= elements[i-1];
        occ_bvec[i] <= next_occ_bvec;

        // If match is true then elements[i-1] will never be unique. :-(
        //
        uniq_at[i] <= match ? n : uniq_at[i-1];

        // Update uniqueness
        //
        uniq_bvec[i] <= !next_occ_bvec || !match && i >= uniq_at[i-1];
        //
        // Future elements[i] is unique if:
        //   - It had been reset.
        //   - It is beyond its unique-at position (uniq_at).

      end

      elements[0] <= element;
      occ_bvec[0] <= 1;
      uniq_at[0] <= match_pos;
      uniq_bvec[0] <= match_pos == 0;
    end

endmodule


module uniq_vector_comb
  #( int we = 10, n = 5, wc = $clog2(n+1) )
   ( output logic [n-1:0] uniq_bvec,
     output logic [wc-1:0] n_match,
     input uwire [we-1:0] element [n-1:0] );

   /// This module is for reference. It should not be part of your solution.
   //  Modify this module only for experimentation.

   // Combinational version.
   always_comb
     for ( int i=0; i<n; i++ ) begin
        automatic logic [wc-1:0] occ = 0;
        for ( int j=0; j<n; j++ ) if ( element[i] == element[j] ) occ++;
        if ( i == 0 ) n_match = occ;
        uniq_bvec[i] = occ == 1;
     end

endmodule



////////////////////////////////////////////////////////////////////////////////
```

/// Testbench Code

```
//
// It is okay to modify the testbench code to facilitate the coding
// and debugging of your modules. Keep in mind that your submission
// will be tested using a different testbench, so on the one hand no
// one will be accused of dishonesty for modifying the testbench
// below. However be sure to restore any changes to make sure that
// your code passes the original testbench.


// cadence translate_off

program reactivate
   (output uwire clk_reactive, output int cycle_reactive,
    input uwire clk, input var int cycle);
   assign clk_reactive = clk;
   assign cycle_reactive = cycle;
endprogram

module testbench;
   localparam int npsets = 4; // This MUST be set to the size of pset.
   // { n we s }
   localparam int pset[npsets][3] =
              '{
                { 4, 4, 0 },
                { 4, 4, 1 },
                { 7, 6, 0 },
                { 7, 6, 1 }
                };

   logic d[npsets:-1]; // Start / Done signals.

   int t_errs_bvec[npsets];
   int t_errs_n_match[npsets];
   int t_n_tests[npsets];

   int t_errs;     // Total number of errors.
   initial begin
      t_errs = 0;
      for ( int i=0; i<npsets; i++ ) begin
         t_errs_bvec[i] = -1;
         t_errs_n_match[i] = -1;
         t_n_tests[i] = -1;
      end
      d[-1] = 1;

      wait( d[npsets-1] );

      for ( int p=0; p<npsets; p++ )
        $write("End of tests n=%2d, s=%0d: %0d bvec errors, %0d n_match errors for %0d tests.\n",
               pset[p][1], pset[p][2],
               t_errs_bvec[p], t_errs_n_match[p], t_n_tests[p]);

   end

   for ( genvar p=0; p<npsets; p++ ) begin
      testbench_n #( .we(pset[p][0]), .n(pset[p][1]), .s(pset[p][2]), .idx(p) )
```

```
      tb( .done(d[p]), .tstart(d[p-1]) );
   end

endmodule


module testbench_n
  #( int n = 12, we = 6, idx = 1, s = 1 )
   ( output logic done, input uwire tstart );

   localparam int wc = $clog2(n+1);

   localparam int n_tests = 100000;
   localparam int cyc_max = n_tests * 2 + 3*n;

   int seed;
   initial seed = 475501;

   function string sample( input string str );
      sample = str[ $dist_uniform( seed, 0, str.len()-1 ) ];
   endfunction

   bit clk;
   int cycle, cycle_limit;
   logic clk_reactive;
   int cycle_reactive;
   reactivate ra(clk_reactive,cycle_reactive,clk,cycle);
   string event_trace;

   initial begin
      clk = 0;
      cycle = 0;
      event_trace = "";

      done = 0;
      cycle_limit = cyc_max;
      wait( tstart );

      fork
         while ( !done ) #1 cycle += clk++;
         wait( cycle >= cycle_limit )
           $write("Exit from clock loop at cycle %0d, limit %0d.  %s\n %s\n",
                  cycle, cycle_limit, "** CYCLE LIMIT EXCEEDED **",
                  event_trace);
      join_any;

      done = 1;
   end


   typedef logic [we-1:0] Digit;
   Digit element_stream[$];
   bit start_stream[$];
   logic [we-1:0] element_stream_x[$];
   int element_most_recent_t[int];
   logic [n-1:0] shadow_uniq_bvec;
   int element_occ[int];
```

```
   uwire [n-1:0] uniq_bvec;
   uwire [wc-1:0] n_match;
   Digit element;
   logic start;

   uniq_vector_seq #(we,n) uvs( uniq_bvec, n_match, element, start, clk );

   int n_err_bvec, n_err_n_match;
   localparam int pre_n = 2;
   localparam int trace_len = 5;
   localparam int n_pre_check = 2 * n; // Number of inputs before testing.
   localparam int start_dups = 5 * n + n_pre_check;

   initial begin
      automatic int n_tests_done = 0;
      int tnum_last_start;
      bit err_bvec_here, err_n_match_here, show_trace;

      n_err_bvec = 0;
      n_err_n_match = 0;
      element = 1;
      start = 1;

      @( negedge clk );
      @( negedge clk );

      start = 0;

      wait( cycle > 2 );

      @( negedge clk );

      $write("\n** Starting tests for n=%0d, input start used = %0s **\n",
             n, s ? "Yes" : "No");

      for ( int tnum=0; tnum<n_tests; tnum++ ) begin

         automatic bit want_match =
           tnum > start_dups && $dist_uniform(seed,0,2) == 0;
         automatic bit want_reset =
           tnum == 0 || s && $dist_uniform(seed,0,1*n) == 0;

         automatic logic [we-1:0] next_element =
           want_match ? element_stream[ {$random} % (n-1) ] : {$random} % 100;
         automatic int n_starts_recent = 0;
         automatic int shadow_n_match = 0;
         bit err_here;

         @( negedge clk );

         if ( element_stream.size() >= n + pre_n ) begin
            automatic int old_element = element_stream.pop_back();
            automatic bit old_start = start_stream.pop_back();
         end

         if ( element_stream.size() >= n
```

```
      && tnum_last_start + n <= tnum )
    element_occ[element_stream[n-1]]--;

  if ( want_reset ) tnum_last_start = tnum;
  element = next_element;
  start = want_reset;

  if ( want_reset ) element_occ.delete();

  element_occ[element]++;
  element_stream.push_front(element);
  start_stream.push_front(start);

  for ( int i=0; i<n; i++ ) begin
     if ( element_stream[i] == element && !n_starts_recent )
       shadow_n_match++;
     shadow_uniq_bvec[i] =
       n_starts_recent || element_occ[element_stream[i]] == 1;
     n_starts_recent += start_stream[i];
  end

  @( posedge clk_reactive );

  if ( tnum < n_pre_check ) continue;

  err_n_match_here = n_match !== shadow_n_match;
  if ( err_n_match_here ) n_err_n_match++;

  err_bvec_here = uniq_bvec !== shadow_uniq_bvec;

  if ( err_bvec_here ) n_err_bvec++;

  err_here = err_bvec_here || err_n_match_here;

  show_trace = tnum > start_dups + n && tnum < start_dups + n + trace_len
    || err_here && n_err_bvec < 5 && n_err_n_match < 5;

  n_tests_done++;

  if ( show_trace ) begin
     $write( "%s, uniq_bvec: t=%0d,  %b",
             err_bvec_here ? "Error" : "Trace", tnum,
             uniq_bvec);
     if ( err_bvec_here )
       $write( "!= %b ( correct )", shadow_uniq_bvec );
     if ( err_n_match_here )
       $write("\nError:  n_match: %0d != %0d (correct)",
              n_match,shadow_n_match);
     $write("\n[");
     for ( int i=n+pre_n-1; i>=n; i-- )
       $write( "%2s%0s%0s ", "", i == n ? "]" : "", i ? "," : "" );
     for ( int i=n-1; i>=0; i-- )
       $write( "%1s%1h%0s%0s ",
               uniq_bvec[i]===shadow_uniq_bvec[i] ? " " : "E",
               uniq_bvec[i],
               i == n ? "]" : "", i ? "," : "" );
     $write(" <-- uniq_bvec\n[");
```

```
            for ( int i=n+pre_n-1; i>=0; i-- )
              $write( "%2d%0s%0s ",
                      element_stream[i], i == n ? "]" : "", i ? "," : "" );
            $write(" <-- Element\n[");
            for ( int i=n+pre_n-1; i>=0; i-- )
              $write( "%2d%0s%0s ",
                      start_stream[i], i == n ? "]" : "", i ? "," : "" );

            $write(" <-- Start\n");
          end

      end

      $write
        ("For n=%0d, s=%0d: done with %0d tests. Errors: %0d bvec, %0d n_match.\n",
         n, s, n_tests, n_err_bvec, n_err_n_match);

      testbench.t_errs_bvec[idx] = n_err_bvec;
      testbench.t_errs_n_match[idx] = n_err_n_match;
      testbench.t_n_tests[idx] = n_tests_done;

      done = 1;

    end

endmodule

// cadence translate_on
```

16 Fall 2022 Solutions

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2022 Homework 1 -- SOLUTION
//

 /// Assignment https://www.ece.lsu.edu/koppel/v/2022/hw01.pdf


`default_nettype none

////////////////////////////////////////////////////////////////////////////////
/// Problem 0
//
 /// Look over the modules and enum below.
//
//


 /// Declare Useful Enumeration Constants.
//
typedef enum
  { Char_space = 32, Char_0 = 48, Char_9 = 57,
    Char_A = 65, Char_Z = 90, Char_a = 97, Char_z = 122 }
  Chars_Special;
//
// See digit_valid_09 to see how these constants can be used.


 /// An ordinary two-input multiplexor.
//
module mux2
  #( int w = 3 )
   ( output uwire [w-1:0] x,
     input uwire s,
     input uwire [w-1:0] a0, a1 );

   assign x = s ? a1 : a0;

endmodule



////////////////////////////////////////////////////////////////////////////////
/// Problem 1
//
 ///  Replace procedural code in atoi1 with modules as described in handout.
 ///
//
//     [✔] atoi1 must instantiate and use a digit_valid_az module.
//     [✔] atoi1 must instantiate and use a char_to_uc module.
//     [✔] atoi1 must instantiate and use mux2 modules.
//     [✔] The completed atoi1 must not have procedural code (always_comb, etc.)
//
```

```
//     [✔] Complete module digit_valid_az.
//     [✔] Complete module char_to_uc.
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//
//     [✔] Don't assume any particular parameter value.
//
//     [✔] Code must be written clearly.
//     [✔] Pay attention to cost and performance.


module char_to_uc( output uwire [7:0] uc, input uwire [7:0] c );
   uwire is_lc = c >= Char_a && c <= Char_z;
   uwire [7:0] uc_if_lc = c - Char_a + Char_A;
   /// SOLUTION
   mux2 #(8) m( uc, is_lc, c, uc_if_lc );
endmodule

module digit_valid_az
  #( int r = 11, vw = $clog2(r) )
   ( output uwire valid, output uwire [vw-1:0] val, input uwire [7:0] char );
   /// SOLUTION
   assign val = 10 + char - Char_A;
   assign valid = char >= Char_A && char < Char_A + r - 10;
endmodule

module atoi1
  #( int r = 32, w = $clog2(r) )
   ( output logic [w-1:0] val,
     output logic is_digit,
     input uwire [7:0] char );

   /// SOLUTION
   logic [w-1:0] val_09, val_az, val_n;
   logic is_09, is_az;

   digit_valid_09 #(r,w) v09( is_09, val_09, char );
   uwire [7:0] char_uc;
   char_to_uc tuc(char_uc,char);
   digit_valid_az #(r,w) vaz( is_az, val_az, char_uc );

   uwire [w-1:0] z = 0;
   mux2 #(w) mval(val_n,is_09,val_az,val_09);
   mux2 #(w) mval0(val,is_digit,z,val_n);

   assign is_digit = is_09 || is_az;

endmodule


module digit_valid_09
  #( int r = 9, vw = $clog2(r) )
   ( output uwire valid, output uwire [vw-1:0] val, input uwire [7:0] char );
```

```
   assign val = char - Char_0;
   assign valid = char >= Char_0 && char <= Char_9 && char < Char_0 + r;
endmodule


 /// Reference Module -- Do Not Modify
//
module atoi1_behavioral
  #( int r = 32, w = $clog2(r) )
   ( output logic [w-1:0] val,
     output logic is_digit,
     input uwire [7:0] char );

   logic [7:0] char_uc;
   logic [w-1:0] val_09, val_az;
   logic is_09, is_az;

   always_comb begin

      char_uc = char >= Char_a && char <= Char_z
        ? char - Char_a + Char_A : char;

      val_09 = char - Char_0;
      val_az = 10 + char_uc - Char_A;

      is_09 = char >= Char_0 && char <= Char_9 && char < Char_0 + r;
      is_az = char_uc >= Char_A && char_uc < Char_A + r - 10;
      is_digit = is_09 || is_az;

      if ( is_09 )       val = val_09;
      else if ( is_az )  val = val_az;
      else               val = 0;

   end

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Testbench Code

// cadence translate_off

module testbench;

   localparam int nradices = 6;
   localparam int radices[nradices] =
              '{ 4, 8, 10, 14, 16, 19 };

   int t_errs;     // Total number of errors.
   initial t_errs = 0;
   final $write("Total number of errors: %0d\n",t_errs);

   uwire d[nradices:-1];    // Start / Done signals.
```

```
   assign d[-1] = 1;  // Initialize first at true.

   // Instantiate a testbench at each size.
   //
   for ( genvar i=0; i<nradices; i++ )
     testbench_r #(radices[i]) t2( .done(d[i]), .tstart(d[i-1]) );

endmodule


module testbench_r
  #( int r = 16 )
   ( output logic done, input uwire tstart );

   localparam int wd = 8;
   localparam int w = $clog2(r**wd);
   localparam int w1 = 2 * $clog2(r);
   localparam int ntests = 500;

   logic [1:0][7:0] str;

   uwire [w1-1:0] val1;
   uwire is_digit1;

   atoi1 #(r,w1) a1( val1, is_digit1, str[0] );

   logic [7:0] non_digit[256];

   function string to_string(input logic [w1-1:0] val);

      automatic string result = "";
      if ( val == 0 ) result = "0";
      while ( val ) begin

         automatic int d = val % r;
         automatic int v = d < 10 ? d + Char_0 : d - 10 + Char_A;
         val = val / r;
         result = { string'(v), result };
      end
      to_string = result;
   endfunction

   initial begin
      automatic int nd_size = 0;
      automatic int rm10 = r > 10 ? 10 : r;
      automatic bit err_silent = 0;
      for ( int i=32; i<128; i++ ) begin
         if ( i >= Char_0 && i < Char_0 + rm10 ) continue;
         if ( i >= Char_A && i < Char_A + r - 10 ) continue;
         if ( i >= Char_a && i < Char_a + r - 10 ) continue;
         non_digit[nd_size++] = i;
      end
      str[1] = Char_space;
```

```
    wait( tstart );

    for ( int tt=0; tt<1; tt++ ) begin

       automatic bit single_char = tt == 0;
       automatic bit space_pad = single_char || tt == 1;
       automatic int nttests = single_char ? 256 : ntests;
       automatic int n_err = 0, n_lerr = 0;
       automatic string ttype =
         single_char ? "Single_Char (SC)"
           : space_pad ? "Space_Pad (SP)" : "General (GE)";
       automatic string abbrev =
         single_char ? "SC" : space_pad ? "SP" : "GE";

    for ( int i=0; i<nttests; i++ ) begin
       automatic int len = single_char ? 1 : 1 + {$random} % wd;
       automatic logic [w1-1:0] sval = 0;
       automatic bit is_09 = i >= Char_0 && i <= Char_9 && i < Char_0 + r;
       automatic int iuc =
         i >= Char_a && i <= Char_z ? i - Char_a + Char_A : i;
       automatic bit is_az = iuc >= Char_A && iuc < Char_A + r - 10;
       automatic bit sis_digit = is_09 || is_az;

       str[0] = i;
       sval = is_09 ? i - Char_0 : is_az ? 10 + iuc - Char_A : 0;

       #1;

       if ( sval !== val1 ) begin
          n_err++;
          if ( !err_silent && n_err < 5 )
            $write("R %2d  Error val 'h%h or %s != %s (correct) for string \"%s\"\n",
                   r, val1, to_string(val1), to_string(sval),
                   string'(str));
       end
       if ( sis_digit !== is_digit1 ) begin
          n_lerr++;
          if ( !err_silent && n_lerr < 5 )
            $write("R %2d  Error is_digit %h != %0d (correct) for string \"%s\"\n",
                   r, is_digit1, sis_digit, string'(str) );
       end

       #1;

    end

       $write("Radix %2d, done with %0d tests, %0d val errors, %0d is_digit errors.\n",
           r, nttests, n_err, n_lerr);

          testbench.t_errs += n_err + n_lerr;
       if ( n_err + n_lerr ) err_silent = 1;
    end
    done = 1;
  end
```

```
endmodule
```

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2022 Homework 2 -- SOLUTION
//

 /// Assignment https://www.ece.lsu.edu/koppel/v/2022/hw02.pdf


////////////////////////////////////////////////////////////////////////////////
/// Problem 0
//
 /// Look over the modules and enum below.
//
//

// Ensure that an omitted type results in an error message.
`default_nettype none

// Module names. (Used by the testbench.)
//
typedef enum { M_proc, M_iter, M_tree, M_iter_sol, M_tree_sol } M_Type;

// Hide the fact that we've memorized ASCII codes.
//
typedef enum
  { Char_0 = 48, Char_9 = 57,
    Char_A = 65, Char_Z = 90, Char_a = 97, Char_z = 122 }
  Chars_Special;


// A function version of atoi1 -- Convert ASCII character to a value.
//
function int atoi1_func( input logic [7:0] char, input int r );
      automatic int char_uc =
        char >= Char_a && char <= Char_z ? char - Char_a + Char_A : char;
      automatic int val_09 = char - Char_0;
      automatic int val_az = 10 + char_uc - Char_A;
      automatic bit is_09 = char>=Char_0 && char <= Char_9 && char < Char_0 + r;
      automatic bit is_az = char_uc >= Char_A && char_uc < Char_A + r - 10;
      atoi1_func = is_09 ? val_09 : is_az ? val_az : -1;
endfunction

module atoi1
  #( int r = 32, w = 10 )
   ( output logic [w-1:0] val,
     output logic is_digit,
     input uwire [7:0] char );
   always_comb begin
      automatic int valr = atoi1_func(char,r);
      is_digit = valr >= 0;
      val = is_digit ? valr : 0;
   end
endmodule

module mux2
  #( int w = 3 )
   ( output uwire [w-1:0] x,
     input uwire s,
     input uwire [w-1:0] a0, a1 );
```

```
   assign x = s ? a1 : a0;
endmodule

module mult_by_c
  #( int w_in = 8, int c = 16, int w_out = w_in+$clog2(c) )
   ( output uwire [w_out-1:0] prod, input uwire [w_in-1:0] a );
   assign prod = a * c;
endmodule

module add
  #( int w = 5 )
   ( output uwire [w-1:0] s, input uwire [w-1:0] a, b );
   assign s = a + b;
endmodule


////////////////////////////////////////////////////////////////////////////////
/// Problem 1
//
 ///  Complete atoi_it so that it computes the string value as follows.
 ///
//
//     [✔] atoi_it must instantiate and use n atoi modules.
//     [✔] atoi_it must instantiate and use mult_by_c modules.
//     [✔] atoi_it must instantiate and use add modules.
//     [✔] Procedural code can be used, but not in place of atoi and mult_by_c.
//
//     [✔] DO NOT use atoi1_func in your solution.
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//
//     [✔] Don't assume any particular parameter value.
//
//     [✔] Code must be written clearly.
//     [✔] Pay attention to cost and performance.


module atoi_it
  #( int r = 11, n = 5, wv = $clog2( r**n ), wd = $clog2(n+1) )
   ( output logic [wv-1:0] val,
     output logic [wd-1:0] nd,
     input uwire [7:0] str [n-1:0] );

   /// SOLUTION

   uwire [wv-1:0] vali[n-1:-1];
   uwire is_valid[n-1:-1];
   uwire [wd-1:0] ndi[n-1:-1];
   assign is_valid[-1] = 1;
   assign ndi[-1] = 0;
   assign vali[-1] = 0;
   assign nd = ndi[n-1];
   assign val = vali[n-1];

   localparam int wcv = $clog2(r);

   for ( genvar i=0; i<n; i++ ) begin

      // Find Value of Digit i
```

```
      //
      uwire [wcv-1:0] valdr;
      uwire is_digit;
      atoi1 #(r,wcv) a( valdr, is_digit, str[i] );

      // Determine if this digit continues a sequence of valid digits
      // starting at str[0].
      //
      assign is_valid[i] = is_digit && is_valid[i-1];

      // Replace value with zero if str[i] is not a digit, or if the
      // string of valid digits has already ended.
      //
      uwire [wcv-1:0] vald = is_valid[i] ? valdr : 0;

      // Multiply (scale) the digit value based on its position in the number.
      //
      uwire [wv-1:0] vals;
      mult_by_c #( .w_in(wcv), .c(r**i), .w_out(wv) ) mc( vals, vald );

      // Add the scaled digit to the value accumulated so far.
      //
      add #(wv) a1( vali[i], vali[i-1], vals );

      // Update the number of digits so far.
      //
      assign ndi[i] = is_valid[i] ? i+1 : ndi[i-1];

   end

endmodule


module atoi_pr
  #( int r = 11, n = 5, wv = $clog2( r**n ), wd = $clog2(n+1) )
   ( output logic [wv-1:0] val,
     output logic [wd-1:0] nd,
     input uwire [7:0] str [n-1:0] );

   /// DO NOT Modify the module. Use it for reference.

   always_comb begin
      val = 0; nd = 0;
      for ( int i=0; i<n; i++ ) begin
         // Get val of current char. If val is < 0 then char is not a digit.
         automatic int dval = atoi1_func(str[i],r);
         if ( dval < 0 ) break;
         val += dval * r**i;
         nd++;
      end
   end

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Problem 2
//
 /// Complete atoi_tr so that it computes the string value as follows.
 ///
```

```
//
//     [✔] atoi_tr must recurisvely instantiate two instances of itself.
//     [✔] atoi_tr must instantiate and use atoi modules.
//     [✔] atoi_tr must instantiate and use mult_by_c modules.
//     [✔] Procedural code can be used, but not in place of atoi and mult_by_c.
//
//     [✔] DO NOT use atoi1_func in your solution.
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//
//     [✔] Don't assume any particular parameter value.
//
//     [✔] Code must be written clearly.
//     [✔] Pay attention to cost and performance.

module atoi_tr
  #( int r = 11, n = 5, wv = $clog2( r**n ), wd = $clog2(n+1) )
   ( output uwire [wv-1:0] val,
     output var logic [wd-1:0] nd,
     input uwire [7:0] str [n-1:0] );

   /// SOLUTION

   if ( n == 1 ) begin

      uwire is_dd;
      uwire [wv-1:0] valr;
      atoi1 #(r,wv) a( valr, is_dd, str[0] );
      assign val = is_dd ? valr : 0;
      assign nd = is_dd; // Note: nd may be more than one bit.

   end else begin

      // Prepare to split the input string into two halves. Note that
      // the hi half may be larger, and so we use nhi to compute the
      // number of bits needed in the value output (vwh) and the
      // number of digits output (dwh).
      //
      localparam int nlo = n/2;
      localparam int nhi = n - nlo;
      localparam int vwh = $clog2( r**nhi );
      localparam int dwh = $clog2( nhi+1 );
      //
      uwire [vwh-1:0] vallo, valhi;
      uwire [dwh-1:0] ndlo, ndhi;

      // Split input string between two recursive instantiations
      //
      atoi_tr #(r,nlo,vwh,dwh) alo( vallo, ndlo, str[nlo-1:0] );
      atoi_tr #(r,nhi,vwh,dwh) ahi( valhi, ndhi, str[n-1:nlo] );

      // Determine whether the hi half of the string may be part
      // of the number.
      //
      uwire hitoo = ndlo == nlo;
      uwire [vwh-1:0] valhid = hitoo ? valhi : 0;

      // Scale the upper half.
      //
      uwire [wv-1:0] valhis;  // VALue HIgh Scaled
```

```
      mult_by_c #(vwh,r**nlo,wv) mc( valhis, valhid );

      assign val = vallo + valhis;
      assign nd = hitoo ? nlo + ndhi : ndlo;

   end

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Testbench Code


// cadence translate_off

module testbench;

   localparam int nnsets = 7;
   localparam int nset[nnsets] = '{ 1, 2, 3, 4,  7, 8, 9 };

   localparam int npsets = 2;
   localparam int pset[npsets] = '{ 10, 16 };

   localparam int nmsets = 2;
   localparam M_Type mset[nmsets] = '{ M_tree, M_iter };

   string mtype_str[M_Type] =
          '{ M_proc:"atoi_pr", M_tree:"atoi_tr", M_iter:"atoi_it",
             M_tree_sol:"a_tr_sol", M_iter_sol:"a_it_sol" };

   int t_errs_len_mod[M_Type];
   int t_errs_val_mod[M_Type];
   int t_errs_len_r[int];
   int t_errs_val_r[int];
   int t_errs_len_n[int];
   int t_errs_val_n[int];

   int t_errs;     // Total number of errors.
   initial t_errs = 0;
   final begin
      for ( int i=0; i<npsets; i++ )
        $write("Total errors for radix %2d: %5d len, %5d val\n",
               pset[i], t_errs_len_r[pset[i]],
               t_errs_val_r[pset[i]]);
      for ( int i=0; i<nnsets; i++ )
        $write("Total errors for string length %2d: %5d len, %5d val\n",
               nset[i], t_errs_len_n[nset[i]],
               t_errs_val_n[nset[i]]);
      for ( int i=0; i<nmsets; i++ )
        $write("Total errors for mod %4s: %5d len, %5d val\n",
               mtype_str[mset[i]], t_errs_len_mod[mset[i]],
               t_errs_val_mod[mset[i]]);
      $write("Total number of errors: %0d\n",t_errs);
   end

   localparam int nsets = nnsets * npsets * nmsets;

   uwire d[nsets:-1]; // Start / Done signals.
   assign d[-1] = 1;  // Initialize first at true.
```

```
   // Instantiate a testbench at each size.
   //
   for ( genvar m=0; m<nmsets; m++ )
     for ( genvar n=0; n<nnsets; n++ )
       for ( genvar i=0; i<npsets; i++ )
         begin
            localparam int idx = m * npsets * nnsets + n * npsets + i;
            testbench_r #(pset[i],nset[n],mset[m])
            t2( .done(d[idx]), .tstart(d[idx-1]) );
       end

endmodule


module testbench_r
  #( int r = 16, n = 3,
     M_Type mtype = M_proc )
   ( output logic done, input uwire tstart );

   localparam int w = $clog2(r**n);
   localparam int ntests = 500;
   localparam int wd = $clog2(n+1);

   uwire [wd-1:0] nd;
   uwire [w-1:0] val;
   logic [7:0] str[n-1:0];

   string mtype_str[M_Type] =
          '{ M_proc:"atoi_pr", M_tree:"atoi_tr", M_iter:"atoi_it",
               M_tree_sol:"a_tr_sol", M_iter_sol:"a_it_sol" };

   case ( mtype )
     M_proc: atoi_pr #(r,n,w) a8( val, nd, str );
     M_tree: atoi_tr #(r,n,w) a8( val, nd, str );
     M_iter: atoi_it #(r,n,w) a8( val, nd, str );
     M_tree_sol: atoi_tr_sol #(r,n,w) a8( val, nd, str );
     M_iter_sol: atoi_it_sol #(r,n,w) a8( val, nd, str );
   endcase

   logic [7:0] non_digit[256];

   function string to_string(input logic [w-1:0] val);

      automatic string result = "";
      if ( val == 0 ) result = "0";
      while ( val ) begin

         automatic int d = val % r;
         automatic int v = d < 10 ? d + Char_0 : d - 10 + Char_A;
         val = val / r;
         result = { string'(v), result };
      end
      to_string = result;
   endfunction

   initial begin
      automatic int nd_size = 0;
      automatic int rm10 = r > 10 ? 10 : r;
      automatic bit err_silent = testbench.t_errs > 10;
      for ( int i=32; i<128; i++ ) begin
```

```
      if ( i >= Char_0 && i < Char_0 + rm10 ) continue;
      if ( i >= Char_A && i < Char_A + r - 10 ) continue;
      if ( i >= Char_a && i < Char_a + r - 10 ) continue;
      non_digit[nd_size++] = i;
   end

   wait( tstart );

   for ( int tt=0; tt<3; tt++ ) begin

      automatic bit single_char = tt == 0;
      automatic bit space_pad = single_char || tt == 1;
      automatic int nttests = single_char ? r : ntests;
      automatic int n_err = 0, n_lerr = 0;
      automatic string ttype =
        single_char ? "Single_Char (SC)"
          : space_pad ? "Space_Pad (SP)" : "General (GE)";
      automatic string abbrev =
        single_char ? "SC" : space_pad ? "SP" : "GE";

      //  $write("Radix %2d, starting %s tests.\n", r, ttype);

   for ( int i=0; i<nttests; i++ ) begin
      automatic int len = single_char ? 1 : 1 + {$random} % n;
      automatic logic [w-1:0] sval = 0;

      for ( int j=0; j<len; j++ ) begin
         automatic int d = {$random} % r;
         automatic int char_a = {$random} % 1 ? Char_A : Char_a;
         if ( d == 0 && j == len - 1 ) d = 1;
         if ( single_char ) d = i;
         str[j] = d < 10 ? Char_0 + d : char_a + d - 10;
         sval += d * r ** j;
      end

      str[len] = space_pad ? " " : non_digit[{$random}%nd_size];

      for ( int j=len+1; j<n; j++ )
        str[j] = space_pad ? 32 : 32 + {$random}%(128-32);

      #1;

      if ( sval !== val ) begin
         n_err++;
         if ( !err_silent && n_err < 5 )
           $write("Mod-%s R-%2d n-%2d Ty-%s  Error val %s != %s (correct) for string \"%s\"\n",
                  mtype_str[mtype], r, n, abbrev, to_string(val), to_string(sval), string'(str));
      end
      if ( !single_char && len !== nd ) begin
         n_lerr++;
         if ( !err_silent && n_lerr < 10 )
           $write("Mod-%s R-%2d n-%2d Ty-%s  Error len %0d != %0d (correct) for string \"%s\"\n",
                  mtype_str[mtype],
                  r, n, abbrev, nd, len, string'(str) );
      end

      #1;

   end

      $write("Mod-%s Radix-%2d n-%2d Ty-%s, done with %0d tests, %0d val errors, %0d len errors.\n",
```

```
                mtype_str[mtype],
              r, n, abbrev, nttests, n_err, n_lerr);

            testbench.t_errs += n_err + n_lerr;
          testbench.t_errs_len_mod[mtype] += n_lerr;
          testbench.t_errs_val_mod[mtype] += n_err;
          testbench.t_errs_len_r[r] += n_lerr;
          testbench.t_errs_val_r[r] += n_err;
          testbench.t_errs_len_n[n] += n_lerr;
          testbench.t_errs_val_n[n] += n_err;
          if ( n_err + n_lerr ) err_silent = 1;
        end
        done = 1;
    end


endmodule

// cadence translate_on
```

LSU EE 4755 Homework 3 Solution Due: 17 Oct 2022, 11:30 CDT

Resources

To help with this assignment review the simple cost model slides and the material in generate statement demo code.

The following problems ask for both inferred hardware and a cost/performance analysis: 2019 Midterm Exam Problem 3c (equality module with shifted inputs), 2021 Midterm Exam Problem 2 (a concentrator for neural network hardware reading sparse weights).

The following are good cost and performance analysis questions (these are the same ones mentioned in the simple model slides): The "find oldest" (big mux) problem covered in class can be found in 2017 Final Exam Problem 3, the knapsack problem hardware covered in class can be found in 2016 Final Exam Problem 2 and 4.

The following are good inferred hardware and optimization problems. Start with 2019 Midterm Exam Problem 1 (a recursively described clz [count leading zeros] module). A problem combining both recursive and iterative generate statements can be found in 202 Midterm Exam Problem 4.

A sequential version of the ASCII-to-value hardware was also assigned in this course. The hardware was described by procedural code and it operated sequentially, so I don't suggest that it specifically be studied for clues on how to solve this assignment.

Problem 1: Compute the cost and delay, using the simple model, of the `atoi1` module (from the solution to Homework 1) instantiated with `r=12`. Base this on a module with reasonable optimizations applied and be sure to account for constants when computing cost and delay.

- Base your analysis of ripple implementations of the adder and magnitude comparison units.
- Show cost.
- Show delay of each output and identify the critical path.
- **Account for constants** when computing cost and delay.

```
module atoi1
  #( int r = 32, w = $clog2(r) )
   ( output logic [w-1:0] val,    output logic is_digit,
     input uwire [7:0] char );

   logic [w-1:0] val_09, val_az, val_n;
   logic is_09, is_az;

   digit_valid_09 #(r,w) v09( is_09, val_09, char );
   uwire [7:0] char_uc;
   char_to_uc tuc(char_uc,char);
   digit_valid_az #(r,w) vaz( is_az, val_az, char_uc );

   uwire [w-1:0] z = 0;
   mux2 #(w) mval(val_n,is_09,val_az,val_09);
   mux2 #(w) mval0(val,is_digit,z,val_n);
```

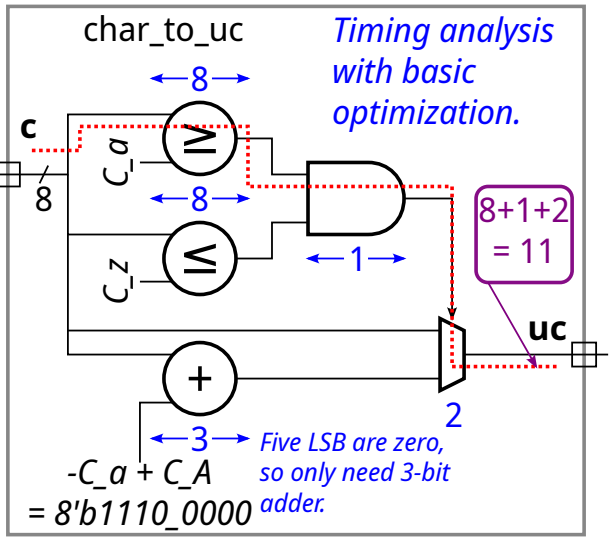
```
   assign is_digit = is_09 || is_az;
endmodule

typedef enum
  { Char_0 = 48, Char_9 = 57, Char_A = 65, Char_Z = 90, Char_a = 97, Char_z = 122 }
  Chars_Special;

module digit_valid_09
  #( int r = 9, vw = $clog2(r) )
   ( output uwire valid, output uwire [vw-1:0] val, input uwire [7:0] char );
   assign val = char - Char_0;
   assign valid = char >= Char_0 && char <= Char_9 && char < Char_0 + r;
endmodule

module char_to_uc( output uwire [7:0] uc, input uwire [7:0] c );
   uwire is_lc = c >= Char_a && c <= Char_z;
   uwire [7:0] uc_if_lc = c - Char_a + Char_A;
   mux2 #(8) m( uc, is_lc, c, uc_if_lc );
endmodule

module digit_valid_az
  #( int r = 11, vw = $clog2(r) )
   ( output uwire valid, output uwire [vw-1:0] val, input uwire [7:0] char );
   assign val = 10 + char - Char_A;
   assign valid = char >= Char_A && char < Char_A + r - 10;
endmodule

module mux2
  #( int w = 3 )
   ( output uwire [w-1:0] x,
     input uwire s,   input uwire [w-1:0] a0, a1 );
   assign x = s ? a1 : a0;
endmodule
```

To start the solution, lets review the cost and delay of common components using the simple model. Those are shown below using symbols $\mathrm{u_c}$ for unit of cost and $\mathrm{u_t}$ for unit of time. For brevity those symbols are omitted in most of the analysis.

A w-Bit Ripple Adder: Cost: $9w\,\mathrm{u_c}$. Delay: $4\,\mathrm{u_t}$ (lsb), $2(w+1)\,\mathrm{u_t}$ (msb).
A w-Bit Ripple Adder with One Constant Input: Cost: $4w\,\mathrm{u_c}$. Delay: $2\,\mathrm{u_t}$ (lsb), $w\,\mathrm{u_t}$ (msb).
A w-Bit Integer Magnitude Unit (Computes $a > b$, $a < b$.) Cost: $4w\,\mathrm{u_c}$. Delay: $2w+1\,\mathrm{u_t}$.
A w-Bit Integer Magnitude Unit with One Constant Input Cost: $w\,\mathrm{u_c}$. Delay: $w\,\mathrm{u_t}$.

In the pages that follow the Verilog descriptions of `atoi` and the modules that it instantiates include comments that show the cost and delay analysis. The words `Cost` and `Delay` are prefixed with abbreviations that indicate the degree of optimization applied. Those abbreviations are:

`N`, No Optimization.
`c0`, Use constant-input cost or delay formulae for the particular device, but make no further optimizations.
`B`, Apply basic optimizations. This includes using the constant-input formulae and making further obvious optimizations.
`G`, Apply good optimizations. These may require careful examination of the computation being performed on that line of Verilog code or an understanding of how the result of the computation is used elsewhere.

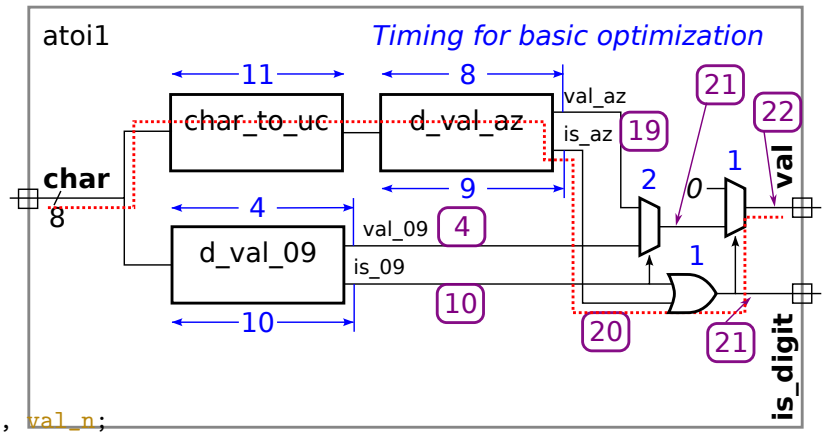



```
module char_to_uc( output uwire [7:0] uc, input uwire [7:0] c );
   uwire is_lc = c >= Char_a  &&  c <= Char_z;
   // n  Cost:   4*8          1   4*8                          = 65
   // B  Cost:   1*8          1   1*8                          = 17
   // B  Delay:  1*8          1   {1*8}                        = 9

   uwire [7:0] uc_if_lc = c - Char_a + Char_A;
   // n  Cost:            9*8
   // c0 Cost:            4*8
   // B  Cost:            4*3      // Five LSB of (-Char_a+Char_A) are zero.
   // B  Delay:           1*3

   mux2 #(8) m( uc, is_lc, c, uc_if_lc );
   // B  Cost:  3*8
   // B  Delay: 2

   /// Module, Basic Optimization
   //  B  Cost  17+12+24 = 53  u_c
   //  B  Critical path: is_lc -> mux : 9 + 2 = 11  u_t

   /// Good Optimization
   //
   uwire is_lc = c[7:5] == 3'b011 && c[4:0] >= 5'b1 && c[4:0] <= 5'b11010;
   // G  Cost:          3         1         1*5     1   1*5   = 15
   // G  Delay:         {2}       1         3       1    {3}  = 5

   assign uc = { char[7:6], char[5] && !is_lc, char[4:0] };
   // G Cost:    0                  1          0
   // G Delay:                      1

   /// Module, Good Optimization
   //  G  Cost  = 15  u_c
   //  G  Delay =  5  u_t
endmodule
```

```
module digit_valid_09
  #( int r = 9, vw = $clog2(r) )
   ( output uwire valid, output uwire [vw-1:0] val, input uwire [7:0] char );
   assign val = char - Char_0;
   // N  Cost:   9*8
   // c0 Cost:   1*8
   // B  Cost:   1*4  (Note: -Char_0 = 8'b11010000, so only adding 4 bits.)
   // B  Delay:  1*4

   assign valid = char >= Char_0  &&  char <= Char_9  &&  char < Char_0 + r;
   // N  Cost:    4*8             1   4*8             1   4*8    = 98
   // B  Cost:    1*8             1   1*8             1   1*8    = 26
   // B  Delay:   1*8             1   {1*8}           1   {1*8}  = 10

  /// Module, Basic Optimization
  //  B  Cost:   4+26 = 30  u_c
  //  B  Delay:       = 10  u_t (Valid output)
  //  B  Delay:       =  4  u_t (Val output)

  /// Good Optimizations
  //
  assign val = char[3:0];  // val can be anything if char isn't 0-9
  //  G Cost:      0
  //  G Delay:     0
  assign valid = char[7:4] == 4'h3   &&  char[3:0] < 10;
  //    Cost:      3                 1   1*4                    = 8
  //    Delay      {2}               1   4                      = 5
  assign valid = char[7:4] == 4'h3   &&  ( !char[3] || !char[2] && !char[1] );
  // G  Cost:      3                 1             1           1  = 6
  // G  Delay:     {2}               1             1           1  = 3

  /// Module, Good Optimization
  //  G  Cost:   = 6  u_c
  //  G  Delay:  = 3  u_t  (Valid output)
  //  G  Delay:  = 0  u_t  (Val output)
endmodule
```

```
module digit_valid_az
  #( int r = 11, vw = $clog2(r) )
   ( output uwire valid, output uwire [vw-1:0] val, input uwire [7:0] char );
   assign val = 10 + char - Char_A;
   // N  Cost:     9*8
   // B  Cost:     4*8                          = 32
   // B  Delay:    1*8                          = 8

   assign valid = char >= Char_A  &&  char < Char_A + r - 10;
   // N  Cost:     4*8             1   4*8
   // B  Cost:     1*8             1   1*8    = 17
   // B  Delay:    1*8             1   {1*8}  = 9

   /// Module, Basic Optimization
   //  B Cost  = 48 u_c
   //  B Delay = 9  u_t

   /// Good Optimizations  (Optimized for r = 12 ).
   assign valid = char[7:2] == 6'b010000  &&  ( char[1:0] == 2'b1 || char[1:0] == 2'b2 )
   // G  Cost:     6                        1     1                 1  1    = 10
   // G  Delay:    3                        1     {1}              {1} {1} = 4
   assign val = { (vw-1)'b101, char[0] };
   // G  Cost:     0
   // G  Delay:    0

   /// Module, Good Optimization
   //  G  Cost  = 10 u_c
   //  G  Delay = 4  u_t  (Valid output)
   //  G  Delay = 0  u_t  (Val output)

endmodule
```

```
module atoi1
  #( int r = 32,
     int w = $clog2(r) )
   ( output uwire [w-1:0] val,
     output uwire is_digit,
     input uwire [7:0] char );

   // Analysis for r = 12 ..
   // .. therefore w = 4

   uwire is_09, is_az;
   uwire [w-1:0] val_09, val_az, val_n;

   digit_valid_09 #(r,w) v09( is_09, val_09, char );
   // B Cost: 30   Delay: 10 (is_09), 4 (val_09)

   uwire [7:0] char_uc;
   char_to_uc tuc(char_uc,char);
   // B Cost: 53   Delay: 11

   digit_valid_az #(r,w) vaz( is_az, val_az, char_uc );
   // B Cost: 48   Delay: 9 (is_az), 8 (val_az)

   uwire [w-1:0] z = 0;
   mux2 #(w) mval( val_n, is_09, val_az, val_09 );
   // B  Cost:   3*w = 3*4 = 12
   // B  Delay:  2

   mux2 #(w) mval0( val, is_digit, z, val_n );
   // N  Cost:   3*w = 3*4 = 12
   // B  Cost:   1*w = 1*4 =  4   (One input is zero.)
   // B  Delay:  1

   assign is_digit = is_09 || is_az;
   // B  Cost 1,  Delay 1

   /// Module, Basic Optimization
   //  B  Cost: 30 + 53 + 48 + 12 + 12 + 1 = 156 u_c
   //  B  A Critical Path: char → char_uc → is_az → is_digit → val
   //  B  Delay: 11 + 9 + 1 + 1 = 22 u_t
   //  B  A Critical Path: char → char_uc → val_az → val_n → val
   //  B  Delay: 11 + 8 + 2 + 1 = 22 u_t


endmodule
```



Problem 2: Appearing further below is the `atoi_it` from the solution to Homework 2.

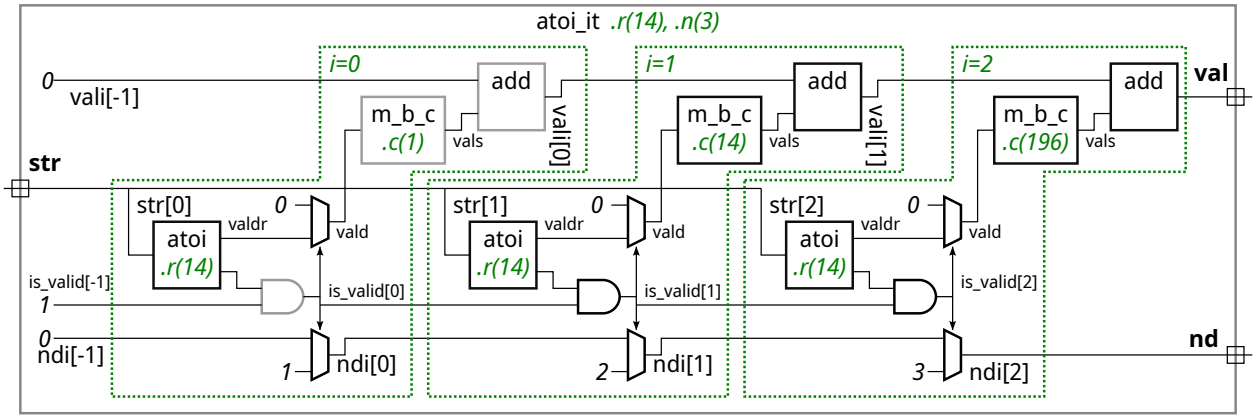
(*a*) Show the hardware inferred for an `atoi_it` module instantiated with `r=14` (yes, radix 14) and `n=3`.

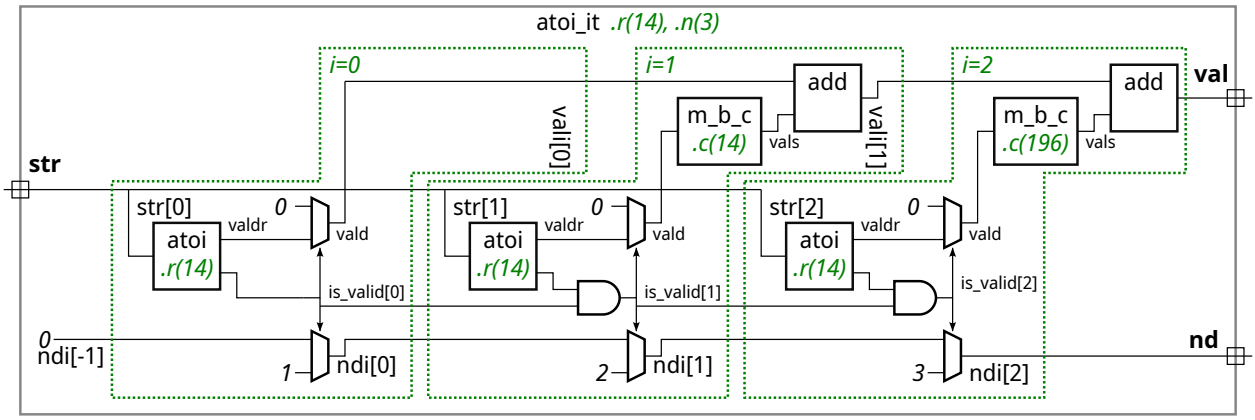
- Show `atoi1`, `mult_by_c`, and `add` instances as modules, do not show what is inside.
- Show the hardware inferred for the operators, such as `&&` and `?:`.
- Do not confuse parameters and ports.
- Omit hardware that does not belong, such as "hardware" to compute values needed at elaboration time.
- Be sure to show the inferred logic. Remember that generate statements describe what happens at elaboration time, not what happens at simulation time nor does it describe operations performed by the hardware.

Solution on the next page.

Solution appears below. The first diagram shows the inferred logic and shows hardware that can easily be removed by optimization in gray. The hardware corresponding to each iteration of the generate loop is shown within a green dotted outline.

In the second diagram the easy optimizations are applied. One easy optimization is the `mult_by_c` module instantiated with `c=1`. Since it would be multiplying by one the output would match the input, and so no hardware is needed. One input to the `add` module on the upper left is zero, so that adder isn't needed. An AND gate is also optimized out.



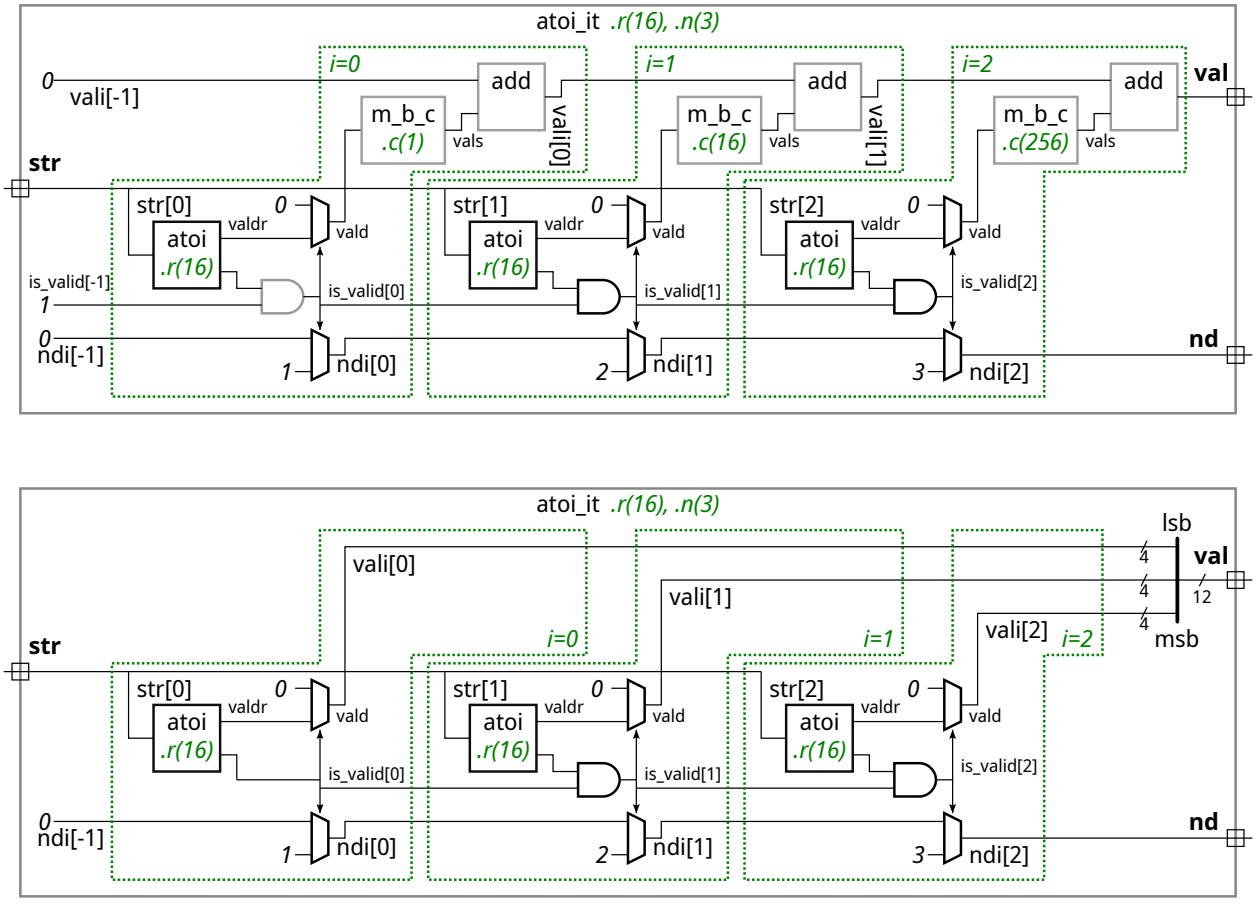


(*b*) Show the hardware inferred for an `atoi_it` module instantiated with `r=16` (hexadecimal this time) and `n=3`, and show the hardware after optimization. Consider the impact of optimization on the `mult_by_c` and `add` modules, which should be considerable since `r` is a power of 2.

The solution appears below. The first diagram shows the inferred logic before optimization. The second diagram shows the optimized hardware, which is substantially less costly since both the `mult_by_c` and `add` modules can be eliminated.

The `mult_by_c` modules can be eliminated because they are multiplying by a power of 2, which can be accomplished simply by re-labeling bit positions. The `add` modules can be eliminated because the two adder inputs will never have a `1` in the same bit position.

For example, consider ASCII input `24'h393635` which should decode to value `12'h965`, in binary `12'b1001_0110_0101`. For this input `vali[0] = 0000_0000_0101` and for `i=1`, `vald=0000_0110_0000`. So all the `i=1` adder really has to do is concatenate the high 8 bits of `vald` with the low 8 bits of `vali[0]`. No addition is necessary. That is shown in the optimized hardware, where each `vald` output is connected directly to their four bit positions in the module output.

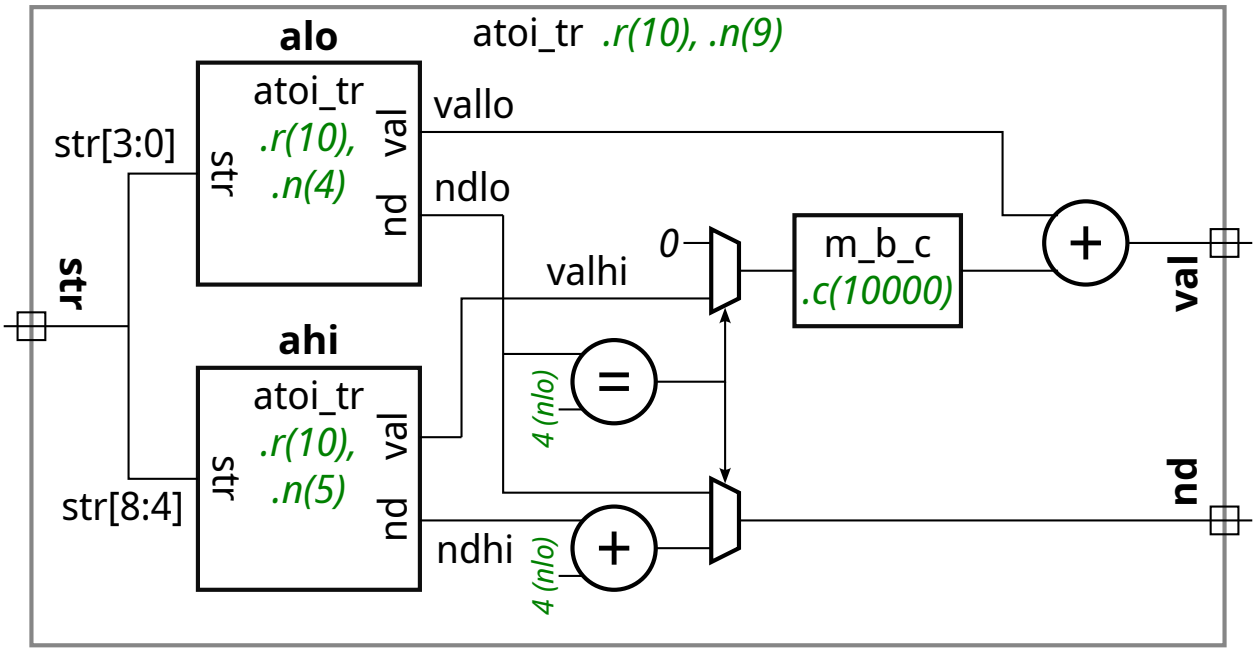


```
module atoi_it
  #( int r = 11, n = 5, wv = $clog2( r**n ), wd = $clog2(n+1) )
   ( output logic [wv-1:0] val,
     output logic [wd-1:0] nd,
     input uwire [7:0] str [n-1:0] );

   uwire [wv-1:0] vali[n-1:-1];
   uwire is_valid[n-1:-1];
   uwire [wd-1:0] ndi[n-1:-1];
   assign is_valid[-1] = 1;
   assign ndi[-1] = 0;
   assign vali[-1] = 0;
   assign nd = ndi[n-1];
   assign val = vali[n-1];

   localparam int wcv = $clog2(r);

   for ( genvar i=0; i<n; i++ ) begin

      // Find Value of Digit i
      //
      uwire [wcv-1:0] valdr;
      uwire is_digit;
      atoi1 #(r,wcv) a( valdr, is_digit, str[i] );

      // Determine if this digit continues a sequence of valid digits
      // starting at str[0].
      //
      assign is_valid[i] = is_digit && is_valid[i-1];

      // Replace value with zero if str[i] is not a digit, or if the
      // string of valid digits has already ended.
      //
      uwire [wcv-1:0] vald = is_valid[i] ? valdr : 0;

      // Multiply (scale) the digit value based on its position in the number.
      //
      uwire [wv-1:0] vals;
      mult_by_c #( .w_in(wcv), .c(r**i), .w_out(wv) ) mc( vals, vald );

      // Add the scaled digit to the value accumulated so far.
      //
      add #(wv) a1( vali[i], vali[i-1], vals );

      // Update the number of digits so far.
      //
      assign ndi[i] = is_valid[i] ? i+1 : ndi[i-1];
   end
endmodule
```

Problem 3: Appearing further below is the `atoi_tr` from the solution to Homework 2. Show the inferred logic for an instantiation with `r=10` and `n=9`.

- Show the logic for one level. That is, show the two instantiations of `atoi_tr`, `alo` and `ahi`, but don't show what is inside of `alo` nor `ahi`.
- Show the `mult_by_c` instantiations as modules, do not show what is inside.
- Show the hardware inferred for the operators, such as `&&` and `?:`.
- Omit hardware that does not belong, such as "hardware" to compute values needed at elaboration time.
- Do not confuse parameters and ports.
- Be sure to show the inferred logic. Remember that generate statements describe what happens at elaboration time, not what happens at simulation time nor does it describe activities performed by the hardware.

Solution appears below. Note that the equality module and adder that operate on the `nd` outputs each have one constant input `ndlo`, which will result in lower-cost and faster hardware. Also note that the value of `nlo` here is 4, and that no hardware is shown computing it. The value is computed by the synthesis (or simulation) program at elaboration time and used to create the module.



```
module atoi_tr
  #( int r = 11, n = 5, wv = $clog2( r**n ), wd = $clog2(n+1) )
   ( output uwire [wv-1:0] val,   output var logic [wd-1:0] nd,
     input uwire [7:0] str [n-1:0] );

   if ( n == 1 ) begin

      uwire is_dd;
      uwire [wv-1:0] valr;
      atoi1 #(r,wv) a( valr, is_dd, str[0] );
      assign val = is_dd ? valr : 0;
      assign nd = is_dd; // Note: nd may be more than one bit.

   end else begin

      // Prepare to split the input string into two halves. Note that
      // the hi half may be larger, and so we use nhi to compute the
      // number of bits needed in the value output (vwh) and the
      // number of digits output (dwh).
      //
      localparam int nlo = n/2;
      localparam int nhi = n - nlo;
      localparam int vwh = $clog2( r**nhi );
      localparam int dwh = $clog2( nhi+1 );
      //
      uwire [vwh-1:0] vallo, valhi;
      uwire [dwh-1:0] ndlo, ndhi;

      // Split input string between two recursive instantiations
      //
      atoi_tr #(r,nlo,vwh,dwh) alo( vallo, ndlo, str[nlo-1:0] );
      atoi_tr #(r,nhi,vwh,dwh) ahi( valhi, ndhi, str[n-1:nlo] );

      // Determine whether the hi half of the string may be part
      // of the number.
      //
      uwire hitoo = ndlo == nlo;
      uwire [vwh-1:0] valhid = hitoo ? valhi : 0;

      // Scale the upper half.
      //
      uwire [wv-1:0] valhis;  // Value High Scaled
      mult_by_c #(vwh,r**nlo,wv) mc( valhis, valhid );

      assign val = vallo + valhis;
      assign nd = hitoo ? nlo + ndhi : ndlo;
   end
endmodule
```

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2022 Homework 4 -- SOLUTION
//

 /// Assignment https://www.ece.lsu.edu/koppel/v/2022/hw04.pdf

typedef enum
  { Char_escape = 128, Char_escape_stop = 200, Char_EOS = 255,
    Char_A = 65, Char_Z = 90, Char_a = 97, Char_z = 122,
    Char_0 = 48, Char_9 = 57,
    Char_space = 32, Char_underscore = 95, Char_cr = 13
    } Chars_Special;

`default_nettype none

////////////////////////////////////////////////////////////////////////////////
/// Problem 1
//
 ///  Complete word_count as described in the handout.
 ///
//
//     [✔] Do not use more adders than are necessary, especially for len_avg.
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//
//     [✔] Don't assume any particular parameter value.
//
//     [✔] Code must be written clearly.
//     [✔] Pay attention to cost and performance.



module word_count
  #( int wl = 5, wn = 6, n_avg_of = 10 )
   ( output logic word_start, word_part, word_ended,
     output logic [wl-1:0] len_word,
     output logic [wn-1:0] num_words,
     output logic [wl-1:0] len_avg,
     input uwire [7:0] char,
     input uwire reset, clk );

   uwire char_az = char >= Char_a && char <= Char_z
         || char >= Char_A && char <= Char_Z;
   uwire char_09 = char >= Char_0 && char <= Char_9;
   uwire char_wd_start = char_az;
   uwire char_wd_part = char_wd_start || char_09 || char == Char_underscore;

   /// SOLUTION

   logic prev_char_wd_part;

   uwire next_word_start = char_wd_start && !prev_char_wd_part;
   uwire next_word_part = word_part && char_wd_part || next_word_start;
   uwire next_word_ended = word_part && !char_wd_part;

   always_ff @( posedge clk ) begin
      prev_char_wd_part <= reset ? 0 : char_wd_part;
      word_start <= reset ? 0 : next_word_start;
      word_part <= reset ? 0 : next_word_part;
      word_ended <= reset ? 0 : next_word_ended;
```

```
   end

   logic [wl-1:0] len_recent[n_avg_of];
   logic [wl+$clog2(n_avg_of):0] len_sum;
   assign len_avg = len_recent[n_avg_of-1] ? len_sum / n_avg_of : 0;

   always_ff @ ( posedge clk ) if ( reset ) begin

      num_words <= 0;
      len_word <= 0;

      for ( int i=0; i<n_avg_of; i++ ) len_recent[i] = 0;
      len_sum = 0;

   end else begin

      len_word <= next_word_start ? 1 : next_word_part ? len_word+1 : len_word;
      num_words <= next_word_ended ? num_words + 1 : num_words;

      if ( next_word_ended ) begin
         len_sum -= len_recent[n_avg_of-1];
         len_sum += len_word;
         for ( int i=n_avg_of-1; i>0; i-- ) len_recent[i] = len_recent[i-1];
         len_recent[0] = len_word;
      end

   end

endmodule

module word_count_blank
  #( int wl = 5, wn = 6, n_avg_of = 10 )
   ( output logic word_start, word_part, word_ended,
     output logic [wl-1:0] len_word,
     output logic [wn-1:0] num_words,
     output logic [wl-1:0] len_avg,
     input uwire [7:0] char,
     input uwire reset, clk );

   uwire char_az = char >= Char_a && char <= Char_z
         || char >= Char_A && char <= Char_Z;
   uwire char_09 = char >= Char_0 && char <= Char_9;
   uwire char_wd_start = char_az;
   uwire char_wd_part = char_wd_start || char_09 || char == Char_underscore;

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Testbench Code
//
// It is okay to modify the testbench code to facilitate the coding
// and debugging of your modules. Keep in mind that your submission
// will be tested using a different testbench, so on the one hand no
// one will be accused of dishonesty for modifying the testbench
// below. However be sure to restore any changes to make sure that
// your code passes the original testbench.


// cadence translate_off

program reactivate
   (output uwire clk_reactive, output int cycle_reactive,
```

```
    input uwire clk, input var int cycle);
   assign clk_reactive = clk;
   assign cycle_reactive = cycle;
endprogram

module testbench;

   localparam int npsets = 3;
   localparam int pset[npsets][2] =
              '{ { 2, 5 }, { 1, 6}, {9, 7 } };


   int n_err_shown;  // Number of times error info printed to console.
   int n_err_sh_nc, n_err_sh_nw, n_err_sh_avg, n_err_sh_state;
   initial begin
      n_err_sh_nc = 0;
      n_err_sh_nw = 0;
      n_err_sh_avg = 0;
      n_err_sh_state = 0;
   end
   int t_errs;       // Total number of errors.
   initial begin t_errs = 0; n_err_shown = 0; end
   final $write("Total number of errors: %0d\n",t_errs);

   uwire d[npsets:-1];    // Start / Done signals.
   assign d[-1] = 1;  // Initialize first at true.

   // Instantiate a testbench at each size.
   //
   for ( genvar i=0; i<npsets; i++ )
     testbench_n #(pset[i][0],pset[i][1]) t2( .done(d[i]), .tstart(d[i-1]) );

endmodule

module testbench_n
  #( int win_sz = 10, wd_len_max = 5 )
   ( output logic done, input uwire tstart );

   localparam int wl = $clog2(wd_len_max+1);
   localparam int wn = $clog2(win_sz) + 5;
   localparam int n_tests = 10000;
   localparam int cyc_max = n_tests * 2;

   // Number of starting trace lines shown.
   localparam int tr_initial_lines = 12;
   // Number of trace lines to show when there is an error.
   localparam int tr_err_context = 5;

   int seed;
   initial seed = 4755;

   function string sample( input string str );
      sample = str[ $dist_uniform( seed, 0, str.len()-1 ) ];
   endfunction

   function string fbit( input logic b, input string s );
      fbit = b === 1 ? s : b === 0 ? "_" : b === 1'bx ? "x" : "z";
   endfunction

   bit clk;
   int cycle, cycle_limit;
   logic clk_reactive;
   int cycle_reactive;
```

```
   reactivate ra(clk_reactive,cycle_reactive,clk,cycle);
   string event_trace;

   initial begin
      clk = 0;
      cycle = 0;
      event_trace = "";

      done = 0;
      cycle_limit = cyc_max;
      wait( tstart );

      fork
         while ( !done ) #1 cycle += clk++;
         wait( cycle >= cycle_limit )
           $write("Exit from clock loop at cycle %0d, limit %0d.  %s\n %s\n",
                  cycle, cycle_limit, "** CYCLE LIMIT EXCEEDED **",
                  event_trace);
      join_any;

      done = 1;
   end

   uwire [wl-1:0] len, lavg;
   uwire [wn-1:0] nw;
   uwire w_start, w_part, w_ended;
   logic [7:0] char;
   logic reset;

   string test_one = "I II III 2not o_wd four cinco a b c d ";
   //  string test_one = "A or bee       ";

   word_count #(wl,wn,win_sz) wd_cnt
     (w_start, w_part, w_ended, len, nw, lavg, char,reset,clk);

   bit char_wd_start[256];
   bit char_wd_part[256];
   string str_wd_start, str_wd_part, str_wd_notstart;
   localparam string str_wd_not = " ,!.-";
   int lens[$];

   initial begin

      automatic logic [wl-1:0] shadow_nc = 0;
      automatic logic [wn-1:0] shadow_nw = 0;
      automatic logic [wl-1:0] shadow_avg = 0;
      automatic int len_sum = 0;
      automatic int n_err_nc = 0, n_err_w_st=0, n_err_w_pa=0, n_err_w_en=0;
      automatic logic shadow_w_st, shadow_w_pa, shadow_w_en;
      automatic int n_err = 0, n_err_lavg = 0, n_err_nw = 0;
      automatic int str_idx = 0;
      automatic string str_win = {10{" "}};
      automatic string test_str_buffer;
      automatic int n_err_pre;
      automatic logic pw_start, pw_part, pw_ended; // State before + edge.
      automatic string tr_recent[$];
      automatic bit need_reset = 0;
      bit in_word, was_in_word, was_word_char;

      for ( int i=0; i<256; i++ )
        begin char_wd_start[i] = 0; char_wd_part[i] = 0; end
      for ( int i=Char_a; i<=Char_z; i++ )
        begin
           char_wd_start[i] = 1; char_wd_part[i] = 1;
```

```
    end
  for ( int i=Char_A; i<=Char_Z; i++ )
    begin
       char_wd_start[i] = 1; char_wd_part[i] = 1;
    end
  for ( int i=Char_0; i<=Char_9; i++ ) char_wd_part[i] = 1;
  char_wd_part[Char_underscore] = 1;

  for ( int i=0; i<256; i++ ) begin
     if ( !char_wd_start[i] && char_wd_part[i] )
       str_wd_notstart = { str_wd_notstart, string'(byte'(i)) };
     if ( char_wd_start[i] )
       str_wd_start = { str_wd_start, string'(byte'(i)) };
     if ( char_wd_part[i] )
       str_wd_part = { str_wd_part, string'(byte'(i)) };
  end

  test_str_buffer = { test_one, test_one };
  str_idx = 0;

  in_word = 0;
  was_in_word = 0;
  was_word_char = 0;
  char = Char_A;
  reset = 1;
  @( posedge clk_reactive ); @( posedge clk_reactive );
  reset = 0;

  for ( int i=0; i<n_tests; i++ ) begin

     automatic int round = i / test_one.len();
     automatic bit do_reset =
       round == 1 && $dist_uniform(seed,1,7) == 1
         || round > 1 && $dist_uniform(seed,1,(wd_len_max+4)/2*win_sz*2)==1;
     automatic bit show_err = 0;

     @( negedge clk );

     if ( str_idx >= test_str_buffer.len() ) begin
        automatic int wd_sz = $dist_uniform(seed,1,wd_len_max);
        automatic int wd_ws = $dist_uniform(seed,1,4);
        automatic bit fake_word = $dist_uniform(seed,1,10) == 1;

        test_str_buffer = "";
        str_idx = 0;

        if ( fake_word )
          test_str_buffer = { test_str_buffer, sample( str_wd_notstart ) };
        else
          test_str_buffer = { test_str_buffer, sample( str_wd_start ) };
        for ( int j=1; j<wd_sz; j++ )
          test_str_buffer = { test_str_buffer, sample( str_wd_part ) };
        for ( int j=0; j<wd_ws; j++ )
          test_str_buffer = { test_str_buffer, sample( str_wd_not ) };

     end

     reset = do_reset;
     char = test_str_buffer[str_idx++];

     if ( round < 1 ) begin

     end else begin
```

```
        end
        str_win = { str_win.substr(1,9), char };

        pw_start = w_start;
        pw_part = w_part;
        pw_ended = w_ended;

        @( posedge clk_reactive );

        if ( do_reset ) begin

           was_in_word = 0;
           was_word_char = 0;
           in_word = 0;
           shadow_w_en = 0;
           shadow_w_pa = 0;
           shadow_w_st = 0;

        end else begin

           was_in_word = in_word;
           shadow_w_st = !was_word_char && char_wd_start[char];
           in_word =
             was_in_word && char_wd_part[char] || shadow_w_st;
           shadow_w_pa = in_word;
           shadow_w_en = was_in_word && !in_word;
           was_word_char = char_wd_part[char];

        end
        if ( do_reset ) begin

           shadow_nc = 0;
           shadow_nw = 0;
           shadow_avg = 0;
           lens.delete();
           len_sum = 0;

        end else if ( was_in_word && in_word ) begin

           shadow_nc++;

        end else if ( shadow_w_en ) begin

           shadow_nw++;
           len_sum += shadow_nc;
           lens.push_front(shadow_nc);
           if ( lens.size() > win_sz )
             len_sum -= lens.pop_back();
           if ( lens.size() == win_sz )
             shadow_avg = len_sum / win_sz;

        end else if ( shadow_w_st ) begin
           shadow_nc = 1;
        end

        n_err_pre = n_err;
        if ( w_start !== shadow_w_st ) begin
           n_err_w_st++;
           n_err++;
        end
        if ( w_part !== shadow_w_pa ) begin
           n_err_w_pa++;
           n_err++;
        end
```

```
    if ( w_ended !== shadow_w_en ) begin
       n_err_w_en++;
       n_err++;
    end
    if ( n_err_pre != n_err ) begin
       if ( testbench.n_err_sh_state++ < 4 ) show_err = 1;
    end

    if ( shadow_nw !== nw )
      begin
         n_err_nw++; n_err++; need_reset = 1;
         if ( testbench.n_err_sh_nw++ < 4 ) show_err = 1;
      end
    if ( shadow_avg !== lavg )
      begin
         n_err_lavg++; n_err++; need_reset = 1;
         if ( testbench.n_err_sh_avg++ < 4 ) show_err = 1;
      end
    if ( shadow_nc !== len ) begin
       n_err_nc++; n_err++;
       if ( testbench.n_err_sh_nc++ < 4 ) show_err = 1;
    end

    begin
       automatic string hd =
         "       W-M    I  Text---->|         SPE L  N A {D}";
       automatic string item =
         $sformatf
           ("Trace %2d-%1d %4d \"%10s\"  %s %s%s%s %s%s%s %1d %2d %1d {%1d}",
            win_sz, wd_len_max, i, str_win,
            do_reset ? "R" : " ",
            fbit(pw_start,"s"), fbit(pw_part,"p"), fbit(pw_ended,"e"),
            fbit(w_start,"S"), fbit(w_part,"P"), fbit(w_ended,"E"),
            len, nw, lavg,
            wd_cnt.char_az
            );

       if ( n_err != n_err_pre )
         item =
           { item,
             $sformatf(" <-Error Correct-> %s%s%s %1d %2d %1d",
                       shadow_w_st ? "S" : "_", shadow_w_pa ? "P" : "_",
                       shadow_w_en ? "E" : "_",
                       shadow_nc, shadow_nw, shadow_avg) };
       if ( i == 0 ) $write("%s\n",hd);
       if ( i < tr_initial_lines )
         $write("%s\n",item);
       else begin
          if ( tr_recent.size() > tr_err_context ) tr_recent.delete(0);
          tr_recent.push_back(item);
       end
    end

    if ( n_err != n_err_pre && show_err ) begin
       while ( tr_recent.size() > 0 )
         $write("%s\n",tr_recent.pop_front());
    end

  end

  $write
    ("Done with n_avg_of=%0d, max wd len=%0d. Errors: st %0d, pa %0d, en %0d, nc %0d, nw %0d, av %0d\n",
     win_sz, wd_len_max,
     n_err_w_st, n_err_w_pa, n_err_w_en,
```

```
         n_err_nc, n_err_nw, n_err_lavg);

      testbench.t_errs += n_err;
      done = 1;


   end

endmodule

// cadence translate_on
```

LSU EE 4755 Homework 5 Solution Due: 22 November 2022

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/2022/hw05.v.html`.

Problem 0: Following instructions at `https://www.ece.lsu.edu/koppel/v/proc.html`, set up your class account, copy the assignment, and run the Verilog simulator and synthesis program on the unmodified homework file, `hw05.v`. Do this early enough so that minor problems (*e.g.*, password doesn't work) are minor problems.

Assignment Background

As we should know the synthesis program, given a Verilog description of a module, writes a design file with an optimized version of the module mapped to the chosen technology. For this assignment the chosen technology is the same Oklahoma University ASIC process we've been using throughout the semester.

An important skill for those writing Verilog descriptions is to estimate the cost and performance of those synthesized modules. In this assignment we'll look at how well the synthesis program handles the different modules we considered for computing the floating-point expression $v_0^2 + v_0 v_1 + v_1^2$. We will consider the combinational, sequential, and pipelined modules covered in class.

A synthesis script will be used to synthesize these modules, plus three arithmetic unit modules, plus additional modules created for the solution to this problem. To complete the assignment the output of the script must be understood and the synthesis script must be modified. The output of the synthesis script is similar to the output of the scripts used in prior assignments, so it should be familiar. Modifying the script will be something new, and might be a challenge for some of you. It is okay to seek help modifying the script from classmates and others, though the solutions to the problems themselves must be completed individually.

Modules

This assignment includes modules for the combinational, sequential, and pipelined implementations of the multi-step computation. They are named `ms_comb`, `ms_seq`, and `ms_pipe`. For comparison the assignment also includes modules containing a single floating-point unit, they are named `try_mult`, `try_add`, and `try_sq` (square).

Four additional modules are provided for experimentation, `m1_func`, `m1_comb`, `m1_seq`, and `m1_pipe`. These modules initially perform the computation $v_0 + v_0 v_1 + v_1^2$, but they can be modified to perform other computations. Module `m1_func` is used by the testbench to obtain a correct value, so modify it first so that it computes the desired computation. Then modify the others that you want to synthesize. (The synthesis program does not care whether a module passes the testbench, but no conclusion can be drawn from the area and delay of module that does not work correctly.)

All of these modules have the same parameters and ports, though not every module uses every port. For example, only `ms_seq` and `ms_pipe` are sequential so that the `clk` and `reset` ports on the others serve no function. These unused ports will be eliminated during optimization so they won't affect cost or timing.

Module Parameters and Floating Point Format

The modules used in this assignment all have the same parameters, these parameters specify the floating-point number format to be used. The first parameter, `wsig`, specifies the number of bits in the significand (fractional part) of the floating point number. The default value is 23, which is the same as an IEEE 754 single (C `float`). The second parameter, `wexp`, is the number of bits in the exponent. The default value is 8, which matches an IEEE single. The third parameter, `ieee`, specifies whether the IEEE floating-point format should be strictly followed. The default value

is 1, which means yes; a 0 means that special cases do not have to be handled correctly. These include `NaN` (not a number) and subnormal values. The size of the floating point number using these parameters is `1+wexp+wsig`, the extra `1` is for the sign bit.

For this assignment all modules are instantiated with `ieee=0`. This is done to explore the fuller range of optimization possibilities and also to reduce the time needed for synthesis.

The sample synthesis runs consider two formats, IEEE single in which `wsig=23` and `wexp=8`, and the ML-friendly BF16 (informally known as brain float) in which `wsig=7` and `wexp=8`. The advantage of BF16 for machine learning is that it is half the size of a single, and with a 7-bit significand, requires half the energy for multiplication than the older 16-bit FP16 format. For us the big advantage is that it takes less time to synthesize than a single.

Testbench

The testbench exercises the six modules, `ms_comb`, `ms_seq`, `ms_pipe`, `m1_comb`, `m1_seq`, and `m1_pipe` instantiated with a significand size of 7 and 23. They should all initially pass. As with other testbenchs in this class, a line will be printed for the first few module errors, and a tally will be provided for each module and size. The testbench uses `ms_func` to determine the correct output of the `ms` modules and `m1_func` to determine the correct output of the `m1` modules. When modifying the `m1` modules be sure to also modify `m1_func` so that the testbench can show you whether your modified modules do what you think they are doing.

The Synthesis Script

As with past assignments, the modules in the assignment file should be synthesized using the script `syn.tcl`. Unlike other assignments, this script will have to be modified.

The synthesis script itself is written in TCL (Tool Control Language, the abbreviation is pronounced tickle) a scripting language chosen by Cadence for scripting their EDA software. (Nowadays Python would be used. If it were up to me it would be Perl. But it's TCL.) Documentation for TCL can be found at `https://tmml.sourceforge.net/doc/tcl/`. This describes TCL, not the functionality needed to run Genus or other tools. For Genus-specific commands see the synthesis documentation linked to `https://www.ece.lsu.edu/koppel/v/ref.html`.

For this assignment it should not be necessary to use new Genus commands, just to change which modules are synthesized and which parameters to instantiate with. For that, one needs only a rudimentary knowledge of TCL, perhaps what can be learned just by looking at `syn.tcl`.

The synthesis script starts by setting some script variables, using the TCL `set` command, and by setting Genus attributes, using the Genus `set_db` command:

```
set verilog_source hw05.v
set syn_level "high"
set spew_file "spew.log"
set report_file "syn-report.log"
set_db syn_global_effort $syn_level
set rpt_chan [open $report_file w]
puts "Synthesizing at effort level \"$syn_level\"\n"
```

As one might guess `syn_level` is the amount of effort used for synthesis. Possible values are `none`, `low`, `medium`, and `high`. These initial lines are followed by the definition of a TCL procedure `syn_mod`, which emits the commands needed to synthesize a module, followed by commands to retrieve the area and delay of the synthesized module. A line of text is written showing the area and delay. It should not be necessary to modify `syn_mod` for this assignment.

Module `syn_mod` is called in a loop nest near the end of the file:

```
# List of combinational modules.
```

```
set mods_comb { ms_comb try_mult try_add try_sq }
set delay_targets { 100 0.1 }
set mods { try_mult try_add try_sq }
set mods { ms_comb ms_seq ms_pipe try_mult try_add try_sq }
set wsigs { 7 14 23 }

foreach delay_target $delay_targets {
    foreach ws $wsigs {
        foreach mod $mods {
            syn_mod $mod $delay_target " $ws 8 0 "
        }
    }
}
```

The loop nest above synthesizes each of the modules listed in `mods` (that's the inner loop). Each of these six modules is synthesized for each significand size found in `wsigs`. These modules are synthesized with each delay constraint in `delay_target`. For the code above there would be a total of $2 \times 6 \times 3$ synthesis runs. That would probably take hours.

The first `set` line writes variable `mods_comb` with a list of combinational modules. This variable must be updated with any new combinational modules that you use. Variable `mods` is set twice, first to a list of the arithmetic modules, then those are replaced with a list of the arithmetic modules and our multi-step modules. (Because of the second assignment the first assignment has no effect.) If one wanted to only synthesize the arithmetic modules one would comment out the second `mods` line. There is no need to use a loop nest. It is possible to write a `syn_mod` call for each synthesis, for example:

```
set delay_targets { 100 }
set wsigs { 7 14 23 }

syn_mod try_mult 5 "7 8 0"
syn_mod try_mult 5 "7 6 0"

# Exit before the loop nest.
close $rpt_chan
quit
foreach delay_target $delay_targets {}
```

The example above does two synthesis runs. The `5` is the delay target and the quoted part are the parameters. (The parameters must be quoted so that they are read as a single argument to `syn_mod`.) In the example above, `try_mult` is synthesized with two exponent sizes, 8 bits and 6 bits, both are synthesized with a delay target of 5 ns.

To synthesize a new module (for example, one you wrote) add the name to one of the `mod` lists, or just use the name on a direct call to `syn_mod` as in the example above. **Iff the module is combinational** add the module to `mods_comb`. Not adding a combinational module to `mods_comb` will result in an error. Adding a sequential module to `mods_comb` will result in incorrect timing.

Synthesis Script Output

The synthesis script `syn.tcl` is run using the command `genus -files syn.tcl`. The run starts with a substantial amount of header output, including warnings, copyright information, and system information. Some is shown below:

```
[cyc.ece.lsu.edu] % genus -files syn.tcl
```

```
2022/11/13 16:52:05 WARNING This OS does not appear to be a Cadence supported Linux configuration.
2022/11/13 16:52:05 For more info, please run CheckSysConf in <cdsRoot/tools.lnx86/bin/checkSysConf <productId>
TMPDIR is being set to /tmp/genus_temp_566634_cyc.ece.lsu.edu_koppel_nvftYI
Cadence Genus(TM) Synthesis Solution.
Copyright 2022 Cadence Design Systems, Inc. All rights reserved worldwide.
Cadence and the Cadence logo are registered trademarks and Genus is a trademark
of Cadence Design Systems, Inc. in the United States and other countries.

[16:52:12.338826] Configured Lic search path (21.01-s002): /apps/linux/cadence/share/license/license.dat:/opt/pgi/license.dat
```

The output of the script proper (as opposed to Genus, the synthesis program) starts with an announcement of the synthesis effort level followed by a table of synthesis results:

```
Synthesizing at effort level "high"

Module Name                          Area   Delay   Delay     Synth
                                            Actual  Target     Time
ms_comb_wsig7_wexp8_ieee0          600190  12.219     0.1 ns    423 s
ms_seq_wsig7_wexp8_ieee0           445400   5.754     0.1 ns    236 s
ms_pipe_wsig7_wexp8_ieee0          797327   5.678     0.1 ns    309 s
ms_comb_wsig14_wexp8_ieee0        1363980  14.391     0.1 ns    707 s
```

Each line of the table shows the result of one synthesis run. The `Module Name` column shows the name of the module followed by the parameter values used in its instantiation. In the sample above three different modules are synthesized, `ms_comb`, `ms_seq`, and `ms_pipe`. Module `ms_comb` is synthesized once with significand of 7 bits and once with a significand of 14 bits.

The `Area` *column* shows the area given by the Genus `report area` command. The units are relative to the OSU technology. *The* `Delay Actual` *column* shows the length of critical path through the module in units of nanoseconds. *The* `Delay Target` *column* shows the delay constraint that the synthesis program was set to meet. In the example above the constraint is 0.1 ns, which means the critical path can be no longer than 0.1 ns. This constraint was intentionally set to an impossibly low value, to determine the minimum delay that the synthesis program could achieve. Normally the delay constraint is set to something achievable, perhaps 4 ns in the example above, and the synthesis program would generate the least expensive design that meets the delay constraint. *The* `Synth Time` *column* shows the wall-clock (elapsed) time needed to perform the synthesis. The wall-clock time is shown to help plan the synthesis runs, it does not directly affect or describe the design itself.

Problem 1: In class we considered three ways of implementing `multi_step`, the modules that computed $v_0^2 + v_0 v_1 + v_1^2$: A combinational version, a sequential version, and a pipelined version. Appearing below are the results from synthesizing these three modules, named `ms_comb`, `ms_seq`, and `ms_pipe`, followed by results of synthesizing modules consisting only of the Chipware floating-point multiplier, adder, and a multiplier with the same value used for both operands. These are synthesized with a large delay constraint, meaning that the cost has been minimized.

Module Name	Area	Delay Actual	Delay Target	Synth Time
ms_comb_wsig23_wexp8_ieee0	1597692	75.142	100.0 ns	229 s
ms_seq_wsig23_wexp8_ieee0	945919	29.324	100.0 ns	111 s
ms_pipe_wsig23_wexp8_ieee0	1866509	28.273	100.0 ns	205 s
try_mult_wsig23_wexp8_ieee0	525991	28.231	100.0 ns	62 s
try_add_wsig23_wexp8_ieee0	339036	27.396	100.0 ns	53 s
try_sq_wsig23_wexp8_ieee0	297753	25.504	100.0 ns	38 s
ms_comb_wsig7_wexp8_ieee0	375767	34.708	100.0 ns	75 s
ms_seq_wsig7_wexp8_ieee0	275858	15.305	100.0 ns	34 s
ms_pipe_wsig7_wexp8_ieee0	526000	14.466	100.0 ns	62 s
try_mult_wsig7_wexp8_ieee0	94274	9.346	100.0 ns	13 s
try_add_wsig7_wexp8_ieee0	140221	14.196	100.0 ns	21 s
try_sq_wsig7_wexp8_ieee0	57802	6.085	100.0 ns	8 s

(*a*) Based on the data above, show the latency and throughput of each module for the 23-bit significand. It might be necessary to look at the module descriptions (Verilog code) to answer this question.

In the discussion below call the value in the `Delay Actual` column of the synthesis results table the *clock period* and let t_c denote its value. For example, for `ms_comb` with the 23-bit significand the clock period is $t_c = 75.142\,\text{ns}$. Also, let L denote latency and θ denote throughput.

Combinational Module, `ms_comb`*:* Latency: $L = t_c = 75.142\,\text{ns}$ and throughput $\theta = \frac{1}{t_c} = \frac{1}{75.142\,\text{ns}}$. The combinational module computes the entire result in one cycle and so the clock period is the latency. It can compute a new result every cycle and so the throughout is the reciprocal of the latency.

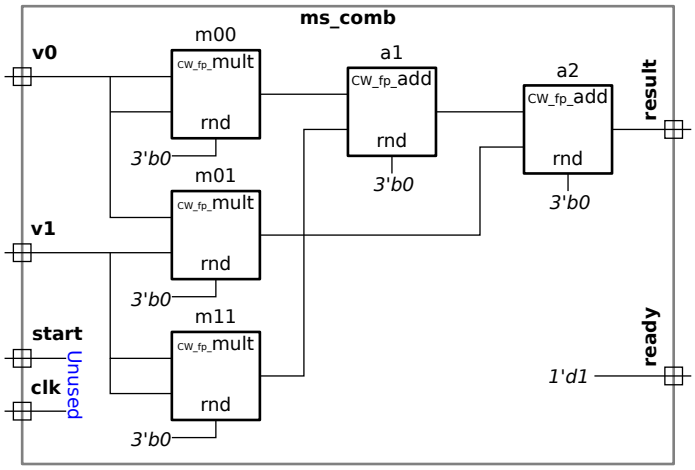
Sequential Module, `ms_seq`*:* Latency: $L = n_c t_c = 5 \times 29.324\,\text{ns} = 146.62\,\text{ns}$, where n_c is the number of cycles needed to compute a result. Throughput: $\theta = \frac{1}{n_c t_c} = \frac{1}{146.62\,\text{ns}}$. The sequential module needs five cycles ($n_c = 5$) to compute a result, so its latency is five times its clock period. Because a new computation cannot start while a computation is in progress the throughput is one over the latency.

Pipelined Module, `ms_pipe`*:* Latency $L = n_s t_c = 4 \times 28.273\,\text{ns}$, where n_s is the number of stages. Throughput $\theta = \frac{1}{t_c} = \frac{1}{28.273\,\text{ns}}$. Like the sequential circuit, the latency of the pipelined unit is the clock period times the number of cycles needed to compute a result. Unlike the sequential circuit, the pipelined circuit can start a new computation every cycle, and so the throughput is the reciprocal of the clock period.

(*b*) For each of the two significand sizes, show that the delay of the three `ms` modules are what one would expect given the delays of the three arithmetic modules.

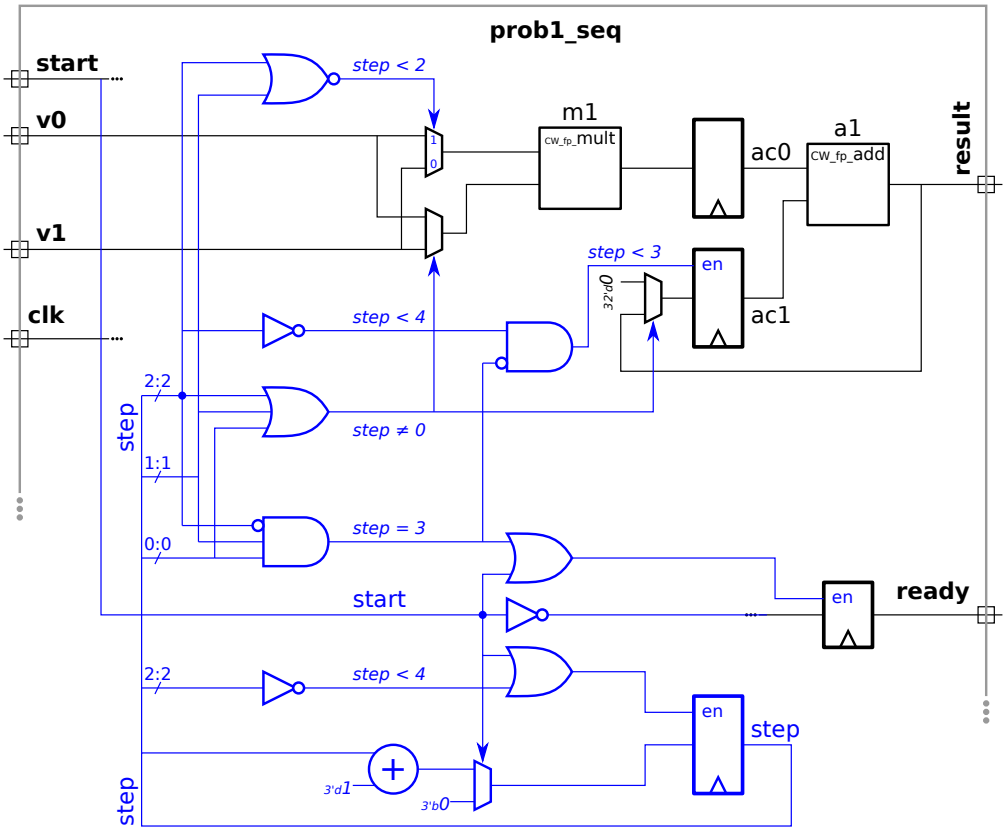
Combinational Module, `ms_comb`*:* To solve this problem one needs to find the critical path through the module. Refer to the Verilog description and the diagram of the inferred hardware below.

```
module ms_comb
  #( int wsig = 23, wexp = 8,
     int ieee = 1,
     int wf = 1 + wexp + wsig )
   ( output uwire [wf-1:0] result,
     output uwire ready,
     input uwire [wf-1:0] v0, v1,
     input uwire start, clk);

   localparam logic [2:0] rm = 0;
   assign ready = 1;

   uwire [7:0] mul_s1, mul_s2, mul_s3;
   uwire [7:0] a_s1, a_s2;
   uwire [wf-1:0] v00, v01, v11, s1;

   CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     m00( .a(v0), .b(v0), .rnd(rm), .z(v00), .status(mul_s1));
   CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     m01( .a(v0), .b(v1), .rnd(rm), .z(v01), .status(mul_s2));
   CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     m11( .a(v1), .b(v1), .rnd(rm), .z(v11), .status(mul_s3));

   CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     a1(.a(v00), .b(v11), .rnd(rm), .z(s1), .status(a_s1));
   CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     a2(.a(s1), .b(v01), .rnd(rm), .z(result), .status(a_s2));
endmodule
```



Based on the timings given in the synthesis results table the critical path goes through `m00`, `a1`, and `a2`. Both `m00` and `m11` compute the square of their inputs, and based on the data in the synthesis results table computing a square takes slightly less time than computing a product, 25.504 ns versus 28.321 ns.

Using the timings from the synthesis table, the delay (critical path) through `ms_comb` is $t_{\text{sq}} + 2t_{\text{add}} = 25.504\,\text{ns} + 2 \times 27.396\,\text{ns} = 80.296\,\text{ns}$. This is about 5 ns longer than the delay for `ms_comb` reported in the synthesis table, 75.142 ns, a difference of only about 6%.



Sequential Module, `ms_seq`*:* The inferred hardware is shown above, taken from the solution to the 2020 final exam (the module name was `prob1_seq` in the exam). For `ms_seq` the critical path cannot pass through both the mutliplier and adder, it must pass through one or the other. In addition to these arithmetic units there is also one multiplexor delay and some logic gates. Assuming that the multiplexor and logic gates' delays are small compared to the arithmetic unit, the critical path will be the larger of the two delays, 28.231 ns for the multiplier and 27.396 ns for the adder. So the clock period would be a bit over 28.231 ns. This is very close to the results from the table, 29.324 ns.
Pipelined Module, `ms_pipe`*:* As with the sequential module, the critical path will be through the arithmetic unit that takes the most time, the multiplier. Unlike the sequential version, there are no multiplexors or logic between the arithmetic units and the pipeline latches, and so we would expect the delay to be even closer to the multiplier delay, 28.231 ns. The reported delay, 28.273 ns is indeed very close.

(*c*) Using the cost of the arithmetic units, show that the cost of `ms_comb` is lower than expected, but the cost of `ms_seq` and `ms_pipe` are about or perhaps a little more than what one would expect.
Combinational Module, `ms_comb`*:* This consists of one multiplier, two square units and two adders. The expected cost is $525991 + 2 \times 339036 + 2 \times 297753 = 1799569$. The reported cost is 1597692, which is lower by 11.2%.
Sequential Module, `ms_seq`*:* This module has one multipler and one adder. Their costs are $525991 + 297753 = 823744$. This estimated cost ignores the cost of registers, multiplexors, and miscellaneous logic. The reported cost is 945919, which is higher, perhaps due to the ignored hardware.
Pipelined Module, `ms_pipe`*:* This module has the same arithmetic units as the combinational module, and so the estimated cost, ignoring registers, would be the same, $525991 + 2 \times 339036 + 2 \times 297753 = 1799569$. The reported cost is 1866509 which is higher. The higher cost is probably due to ignoring the cost of registers.

Problem 2: It is welcome that the cost of `ms_comb` is lower than what one would expect based on the cost of the arithmetic units. There are several possible reasons for this, for example the synthesis program may be simplifying the two adders used in computations such as $a + b + c$ or it may be sharing hardware used to process the common b operand in expressions like $a \times b$ and $b \times c$, or perhaps it may even be transforming $v_0^2 + v_0 v_1 + v_1^2$ into $(v_0 + v_1)^2 - v_0 v_1$. Or maybe the costs for the arithmetic units shown in the table are higher than they should be.

Perform a set of synthesis runs to provide evidence for a reason that `ms_comb` cost less than its constituent parts. Consider the possible reasons given above, or one of your own. These synthesis runs can operate on one of the existing modules, a slightly modified version of the modules, or something wholly different. The modules `m1_comb`, `m1_seq`, `m1_pipe` can be used for experimentation. See the Modules section above.

Describe the results of these experiments and how they convincingly support a particular reason for the lower cost. Data from a single synthesis run, or a series of very similar runs will not be considered convincing.

The Verilog file for this assignment will be collected, but submit the answers to this question on paper or by E-mail. Please E-mail PDF files. Sending word processor source files as a final product is unprofessional, even if they are TEX files.

In your writeup:

- Indicate how you believe the synthesis program is optimizing `ms_comb`.
- Describe the modules you synthesized to come to this conclusion, and the results of synthesis. Most credit will be given for this part of the assignment.
- Explain why your experiments show that the lower cost was not due to other optimizations.

Based on the experiments described below, it appears that the synthesis program can significantly reduce the cost of computations of the form $a^2 + b^2$ computed using the ChipWare FP arithmetic modules. The optimization is not applied to similar computations such as $(a + c)^2 + b^2$.

To determine why the cost of `ms_comb` was more than 11% less than the estimated cost, a number of new modules were simulated. The modules were designed to test various hypotheses, including those suggested in the problem. The modules' names all start with `m1_`, followed by an abbreviation that may suggest what it does. (In the table of synthesis results the name is appended with the parameter values used.) For example, `m1_a3` is a module that computes $a+b+c$. Each module was tested for correctness by updating `m1_functional` so that it computes the same value as the test module. A wrapper module, `m1_comb`, provides a third input for modules that take three data inputs, such as `m1_a3`. The third input value is just `v0*v1`, so `m1_functional` uses `v0*v1` in places where `v2` might go. All of the tested modules were combinational. The synthesis script output shown below (near the end of the solution) is for runs using these modules.

The Verilog code used for these experiments can be found at
`https://www.ece.lsu.edu/koppel/v/2022/hw05-sol.v.html` and the synthesis script is at
`https://www.ece.lsu.edu/koppel/v/2022/syn-sol.tcl.html`.

Appearing below (on the next page) is the `m1_a3` module, its wrapper, and `m1_functional`.

```
/// This module is synthesized.
module m1_a3
  #( int wsig = 23, wexp = 8, ieee = 1, wf = 1 + wexp + wsig )
   ( output uwire [wf-1:0] result,       output uwire ready,
     input uwire [wf-1:0] v0, v1, v2,    input uwire start, clk);

   localparam logic [2:0] rm = 0; // Rounding Mode

   uwire [7:0] mul_s1, mul_s2, mul_s3, a_s1, a_s2;
   uwire [wf-1:0] v00, v01, v11, s1;

   CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     a1(.a(v0), .b(v1), .rnd(rm), .z(s1), .status(a_s1));
   CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     a2(.a(s1), .b(v2), .rnd(rm), .z(result), .status(a_s2));

   assign ready = 1;
endmodule

/// This module is simulated
module m1_comb
  #( int wsig = 23, wexp = 8, ieee = 1, wf = 1 + wexp + wsig )
   ( output uwire [wf-1:0] result,    output uwire ready,
     input uwire [wf-1:0] v0, v1,     input uwire start, clk);

   localparam logic [2:0] rm = 0; // Rounding Mode
   uwire [wf-1:0]  v01;
   uwire [7:0] mul_s2;

   // Generate a third input for m1_a3.
   CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     m01( .a(v0), .b(v1), .rnd(rm), .z(v01), .status(mul_s2));

   m1_a3 #( .wsig(wsig), .wexp(wexp), .ieee(ieee) )
   a3( result, ready, v0, v1, v01, start, clk );
endmodule

// cadence translate_off
module m1_functional
  ( output real mag, input real v0, v1 );
   // The testbench uses this module to test the others, so set
   // the computation to match the others.
   localparam string name = "A3 Func";
   // Note: The third value is v0*v1.
   always_comb mag = v0 + v1 + v0 * v1;
endmodule
// cadence translate_on
```

The results of each experiment are described below. The value inputs to the modules are called v_0, v_1, and v_2. The original multi-step modules only had two data inputs, v_0 and v_1. The third input, v_2, is set to $v_0 v_1$ by the wrapper module, `m1_comb`, for testing purposes. The synthesis program operates on modules such as `m1_a3` and so to it the value on third input, v_2, is unrelated to the other two values.

In the discussion below let c_a, c_m, and c_s denote the cost of the adder, multiplier, and square unit. Those costs are $c_m = 525991$, $c_a = 339036$, and $c_s = 297753$ for the 23-bit significand and $c_m = 94274$, $c_a = 140221$, and $c_s = 57802$ for the 7-bit significand.

Module `m1_a3`*: Computes* $v_0 + v_1 + v_2$

To test for any benefit of computing $a + b + c$ use a module that computes this sum, `m1_a3`. The expected cost is two adders, $3c_a$, which is 280442 for 7 bits and 678072 for 23 bits. The synthesized costs are 278532 and 668518, respectively or .68% and 1.41% lower than estimated. So there is not much optimization benefit from combining two adders.

Module `m1_mad`*: Computes* $v_0 * v_1 + v_2$

To test whether adder and multiplier hardware is shared, try a module that computes $v_0 \times v_1 + v - 2$, called `m1_mad`. The expected cost is $c_m + c + a$ or 234495 for 7 bits and 865027 for 23 bits. The synthesized hardware is just 2.58% and 1.48% less costly than the estimate, not enough to explain `ms_comb`.

Module `m1_mm`*: Computes* $v_0 * v_1$ *and* $v_0 * v_2$

Perhaps two multipliers that have a common multiplier can share some hardware. Module `m1_mm` tests that by using v_0 in both multiplies. This module has two outputs, one for each product. So the estimated cost is $2c_m$: 188548 and 1051982 for the 7- and 23-bit versions. The synthesized cost is just 1.76% and .59% less than the estimates.

Module `m1_comb_v3`*: Computes* $v_0^2 + v_0 v_2 + v_1^2$

To rule out whether the cost reduction is due to an algebraic transformation, a version of `ms_comb` which has three value inputs was tried. The new value, v_2, replaces v_1 in the $v_0 v_1$ term. The estimated cost is $2c_s + c_m + 2c_a$, the same as the `ms_comb` estimate. The synthesized costs are 22.24% and 10.79% lower than the estimated costs, which means that the synthesis program is not doing an algebraic transformation that depends on the middle term, $v_0 v_1$, sharing a variable with the other two.

Module `m1_comb_sos`*: Computes* $v_0^2 + v_1^2$

Perhaps there's something special about a sum of squares. The estimated cost is $2c_s + c_a$, or 255825 and 934542 for the 7- and 23-bit versions. The synthesized costs are substantially lower, 38.3% and 8.61%. The fact that the benefit is larger for the smaller significand suggests that the savings is with the handling of the exponents, which are eight bits in both versions.

Module `m1_comb_sop`*: Computes* $v_0 v_2 + v_1 v_3$

Are squares special? To rule that out a module computing a sum of two products was tried. This module has four value inputs. The estimated cost is $2c_m + c_1$ or 328769 and 1391018. The synthesized costs are 1.99% and 1.58% less, suggesting that there is something special about a sum of squares.

Module `m1_comb_ssp`*: Computes* $v_0^2 + v_1 v_2$

Perhaps one square can be optimized, `m1_comb_ssp` tests that. The expected cost is $c_m + c_s + c + a$ or 292297 and 1162780. The synthesized costs are 7.48% and 2.31% less, so there is some benefit to one square, but not nearly as much as the benefit from both adder inputs being squares.

Module `m1_comb_alt`*: Computes* $(v_0^2 + v_0 v_1) + v_1^2$

Finally, just to be sure, re-do `ms_comb` so the two squares are not operands of the same adder. The expected cost is $2c_s + c_m + 2c_a$ or 441037 and 1726894. The synthesized costs are lower, 10.05% and 4.04%, suggesting that there is some benefit of using a square input to an adder, but that the benefit is substantially larger when both inputs are a square.

Synthesis Data on Next Page

Module Name	Area	Delay Actual	Delay Target	Synth Time
m1_a3_wsig7_wexp8_ieee0	278532	28.162	100.0 ns	54 s
m1_a3_wsig23_wexp8_ieee0	668518	54.005	100.0 ns	117 s
m1_mad_wsig7_wexp8_ieee0	228453	23.283	100.0 ns	36 s
m1_mad_wsig23_wexp8_ieee0	852191	53.147	100.0 ns	114 s
m1_mm_wsig7_wexp8_ieee0	185236	9.346	100.0 ns	20 s
m1_mm_wsig23_wexp8_ieee0	1045808	28.231	100.0 ns	80 s
m1_comb_v3_wsig7_wexp8_ieee0	381271	34.077	100.0 ns	79 s
m1_comb_v3_wsig23_wexp8_ieee0	1605466	74.753	100.0 ns	276 s
m1_comb_sos_wsig7_wexp8_ieee0	157807	19.544	100.0 ns	34 s
m1_comb_sos_wsig23_wexp8_ieee0	854120	48.375	100.0 ns	128 s
m1_comb_sop_wsig7_wexp8_ieee0	322223	23.763	100.0 ns	48 s
m1_comb_sop_wsig23_wexp8_ieee0	1369003	53.049	100.0 ns	169 s
m1_comb_ssp_wsig7_wexp8_ieee0	270427	23.781	100.0 ns	44 s
m1_comb_ssp_wsig23_wexp8_ieee0	1135920	53.306	100.0 ns	152 s
m1_comb_alt_wsig7_wexp8_ieee0	441037	38.112	100.0 ns	103 s
m1_comb_alt_wsig23_wexp8_ieee0	1726894	80.177	100.0 ns	281 s

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2022 Homework 5 -- SOLUTION
//

 /// Assignment https://www.ece.lsu.edu/koppel/v/2022/hw05.pdf
 /// Solution writeup https://www.ece.lsu.edu/koppel/v/2022/hw05_sol.pdf

`default_nettype none

////////////////////////////////////////////////////////////////////////////////
/// All Problems
//
 ///  Arithmetic-Unit-Only Modules
//
//    These modules have a single arithmetic module.
//
//    Use these to estimate the cost of the multi-step complex modules.
//
//    The ports and parameters match the multi-step for convenience.

module try_mult
  #( int wsig = 23, wexp = 8, ieee = 1, wf = 1 + wexp + wsig )
   ( output uwire [wf-1:0] result,
     output uwire ready,
     input uwire [wf-1:0] v0, v1,
     input uwire start, clk);

   localparam logic [2:0] rm = 0; // Rounding Mode
   uwire [7:0] mul_s1;

   CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     m00( .a(v0), .b(v1), .rnd(rm), .z(result), .status(mul_s1));

   assign ready = 1;

endmodule

module try_sq
  #( int wsig = 23, wexp = 8, ieee = 1, wf = 1 + wexp + wsig )
   ( output uwire [wf-1:0] result,
     output uwire ready,
     input uwire [wf-1:0] v0,
     input uwire start, clk);

   try_mult #(wsig,wexp,ieee) tm( result, ready, v0, v0, start, clk);

endmodule

module try_add
  #( int wsig = 23, wexp = 8, ieee = 1, wf = 1 + wexp + wsig )
   ( output uwire [wf-1:0] result,
     output uwire ready,
     input uwire [wf-1:0] v0, v1,
     input uwire start, clk );

   localparam logic [2:0] rm = 0; // Rounding Mode
   uwire [7:0] add_s1;
```

```
   CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     a00( .a(v0), .b(v1), .rnd(rm), .z(result), .status(add_s1));

   assign ready = 1;

endmodule


 ///  Multi-Step Modules
//
//    These compute the function in three different ways.
//
//    Do not modify these modules.
//    Modify the m1 modules instead.


// cadence translate_off
module ms_functional
  ( output real mag, input real v0, v1 );

   localparam string name = "Func";

   always_comb mag = v0 * v0 + v0 * v1 + v1 * v1;

endmodule
// cadence translate_on


module ms_comb
  #( int wsig = 23, wexp = 8, ieee = 1, wf = 1 + wexp + wsig )
   ( output uwire [wf-1:0] result,
     output uwire ready,
     input uwire [wf-1:0] v0, v1,
     input uwire start, clk);

   // cadence translate_off
   localparam string name = "Comb";
   // cadence translate_on

   localparam int nstages = 1;
   localparam logic [2:0] rm = 0; // Rounding Mode

   uwire [7:0] mul_s1, mul_s2, mul_s3, a_s1, a_s2;
   uwire [wf-1:0] v00, v01, v11, s1;

   CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     m00( .a(v0), .b(v0), .rnd(rm), .z(v00), .status(mul_s1));
   CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     m01( .a(v0), .b(v1), .rnd(rm), .z(v01), .status(mul_s2));
   CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     m11( .a(v1), .b(v1), .rnd(rm), .z(v11), .status(mul_s3));

   CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     a1(.a(v00), .b(v11), .rnd(rm), .z(s1), .status(a_s1));
   CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     a2(.a(s1), .b(v01), .rnd(rm), .z(result), .status(a_s2));

   assign ready = 1;
```

```
endmodule


module ms_seq
  #( int wsig = 23, wexp = 8, ieee = 1, wf = 1 + wexp + wsig )
   ( output logic [wf-1:0] result,  output logic ready,
     input uwire [wf-1:0] v0, v1,   input uwire start, clk );

   // cadence translate_off
   localparam string name = "Seq";
   // cadence translate_on

   uwire [7:0] mul_s, add_s; // Operation status. Ignored.
   uwire [wf-1:0] mul_a, mul_b, add_a, add_b, prod, sum;

   logic [2:0]  step;
   logic [wf-1:0] ac0, ac1;

   localparam int last_step = 4;

   always_ff @( posedge clk )
     if ( start ) step <= 0;
     else if ( step < last_step ) step <= step + 1;

   localparam logic [2:0] rm = 0; // Rounding Mode
   CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     m1( .z(prod), .a(mul_a), .b(mul_b), .rnd(rm), .status(mul_s));
   CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     a1( .z(sum),  .a(add_a), .b(add_b), .rnd(rm), .status(add_s));

   assign mul_a = step < 2  ? v0 : v1;
   assign mul_b = step == 0 ? v0 : v1;
   assign add_a = ac0,  add_b = ac1;

   always_ff @( posedge clk )
     begin
        ac0 <= prod;
        if ( step < 3 ) ac1 <= step ? sum : 0;
        if ( start ) ready <= 0; else if ( step == last_step-1 ) ready <= 1;
     end

   assign result = sum;

endmodule

module ms_pipe
  #( int wsig = 23, wexp = 8, ieee = 1, wf = 1 + wexp + wsig )
   ( output uwire [wf-1:0] result,
     output uwire ready,
     input uwire [wf-1:0] v0, v1,
     input uwire start, clk);

   // cadence translate_off
   localparam string name = "Pipe";
   // cadence translate_on

   localparam int nstages = 4;
   localparam logic [2:0] rm = 0; // Rounding Mode
```

```
   uwire [7:0] mul_s1, mul_s2, mul_s3, a_s1, a_s2;
   uwire [wf-1:0] v00, v01, v11, s1, s2;
   logic [wf-1:0] pl_1_v00, pl_1_v01, pl_1_v11;
   logic [wf-1:0] pl_2_v0001, pl_2_v11;
   logic [wf-1:0] pl_3_sum;
   logic pl_1_occ, pl_2_occ, pl_3_occ;

   CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     m00( .z(v00), .a(v0), .b(v0), .rnd(rm), .status(mul_s1));
   CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     m01( .z(v01), .a(v0), .b(v1), .rnd(rm), .status(mul_s2));
   CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     m11( .z(v11), .a(v1), .b(v1), .rnd(rm), .status(mul_s3));

   CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     a1(.z(s1), .a(pl_1_v00),   .b(pl_1_v01), .rnd(rm), .status(a_s1));
   CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     a2(.z(s2), .a(pl_2_v0001), .b(pl_2_v11), .rnd(rm), .status(a_s2));

   assign ready = pl_3_occ;
   assign result = pl_3_sum;

   always_ff @( posedge clk ) begin

      pl_1_v00 <= v00;
      pl_1_v01 <= v01;
      pl_1_v11 <= v11;
      pl_1_occ <= start;

      pl_2_v0001 <= s1;
      pl_2_v11 <= pl_1_v11;
      pl_2_occ <= pl_1_occ;

      pl_3_sum <= s2;
      pl_3_occ <= pl_2_occ;

   end

endmodule


 ///  Experimentation Modules
//
//    These compute a different function in three different ways.
//
//    Modify these modules.
//



// cadence translate_off
module m1_functional
  ( output real mag,
    input real v0, v1 );

   // The testbench uses this module to test the others, so set
   // the computation to match the others.
```

```
    localparam string name = "A3 Func";
    //  always_comb mag = v0 + v0 * v1 + v1 * v1;
    // always_comb mag = v0 * v1 + v1 + v0;
    //  always_comb mag = v0 * v1 + v0 * v1 * v0 ;
    //  always_comb mag = v0 * v0 + v0 * v1 * v0 + v1 * v1;
    //  always_comb mag = v0 * v0 + v0 * v1;
    always_comb mag = v0 * v0 + v0 * v1 + v1 * v1;


endmodule
// cadence translate_on

module m1_comb_alt
  #( int wsig = 23, wexp = 8, ieee = 1, wf = 1 + wexp + wsig )
   ( output uwire [wf-1:0] result,
     output uwire ready,
     input uwire [wf-1:0] v0, v1,
     input uwire start, clk);

    // cadence translate_off
    localparam string name = "Alt Comb";
    // cadence translate_on

    localparam int nstages = 1;
    localparam logic [2:0] rm = 0; // Rounding Mode

    uwire [7:0] mul_s1, mul_s2, mul_s3, a_s1, a_s2;
    uwire [wf-1:0] v00, v01, v11, s1;

    CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
      m00( .a(v0), .b(v0), .rnd(rm), .z(v00), .status(mul_s1));
    CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
      m01( .a(v0), .b(v1), .rnd(rm), .z(v01), .status(mul_s2));
    CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
      m11( .a(v1), .b(v1), .rnd(rm), .z(v11), .status(mul_s3));

    CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
      a1(.a(v00), .b(v01), .rnd(rm), .z(s1), .status(a_s1));
    CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
      a2(.a(s1), .b(v11), .rnd(rm), .z(result), .status(a_s2));

    assign ready = 1;

endmodule


module m1_comb_ssp
  #( int wsig = 23, wexp = 8, ieee = 1, wf = 1 + wexp + wsig )
   ( output uwire [wf-1:0] result,
     output uwire ready,
     input uwire [wf-1:0] v0, v1, v2,
     input uwire start, clk);

    // cadence translate_off
    localparam string name = "Comb";
    // cadence translate_on

    localparam int nstages = 1;
    localparam logic [2:0] rm = 0; // Rounding Mode
```

```
    uwire [7:0] mul_s1, mul_s2, mul_s3, a_s1, a_s2;
    uwire [wf-1:0] v00, v01, v11, s1;

    CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
      m00( .a(v0), .b(v0), .rnd(rm), .z(v00), .status(mul_s1));
    CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
      m11( .a(v1), .b(v2), .rnd(rm), .z(v11), .status(mul_s3));

    CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
      a1(.a(v00), .b(v11), .rnd(rm), .z(result), .status(a_s1));

    assign ready = 1;

endmodule

module m1_comb_sop
  #( int wsig = 23, wexp = 8, ieee = 1, wf = 1 + wexp + wsig )
   ( output uwire [wf-1:0] result,
     output uwire ready,
     input uwire [wf-1:0] v0, v1, v2, v3,
     input uwire start, clk);

    // cadence translate_off
    localparam string name = "Comb";
    // cadence translate_on

    localparam int nstages = 1;
    localparam logic [2:0] rm = 0; // Rounding Mode

    uwire [7:0] mul_s1, mul_s2, mul_s3, a_s1, a_s2;
    uwire [wf-1:0] v00, v01, v11, s1;

    CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
      m00( .a(v0), .b(v2), .rnd(rm), .z(v00), .status(mul_s1));
    CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
      m11( .a(v1), .b(v3), .rnd(rm), .z(v11), .status(mul_s3));

    CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
      a1(.a(v00), .b(v11), .rnd(rm), .z(result), .status(a_s1));

    assign ready = 1;

endmodule

module m1_comb_sos
  #( int wsig = 23, wexp = 8, ieee = 1, wf = 1 + wexp + wsig )
   ( output uwire [wf-1:0] result,
     output uwire ready,
     input uwire [wf-1:0] v0, v1,
     input uwire start, clk);

    // cadence translate_off
    localparam string name = "Comb";
    // cadence translate_on

    localparam int nstages = 1;
    localparam logic [2:0] rm = 0; // Rounding Mode
```

```
    uwire [7:0] mul_s1, mul_s2, mul_s3, a_s1, a_s2;
    uwire [wf-1:0] v00, v01, v11, s1;

    CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
      m00( .a(v0), .b(v0), .rnd(rm), .z(v00), .status(mul_s1));
    CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
      m11( .a(v1), .b(v1), .rnd(rm), .z(v11), .status(mul_s3));

    CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
      a1(.a(v00), .b(v11), .rnd(rm), .z(result), .status(a_s1));

    assign ready = 1;

endmodule

module m1_comb_v3
  #( int wsig = 23, wexp = 8, ieee = 1, wf = 1 + wexp + wsig )
   ( output uwire [wf-1:0] result,
     output uwire ready,
     input uwire [wf-1:0] v0, v1, v2,
     input uwire start, clk);

    // cadence translate_off
    localparam string name = "Comb";
    // cadence translate_on

    localparam int nstages = 1;
    localparam logic [2:0] rm = 0; // Rounding Mode

    uwire [7:0] mul_s1, mul_s2, mul_s3, a_s1, a_s2;
    uwire [wf-1:0] v00, v01, v11, s1;

    CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
      m00( .a(v0), .b(v0), .rnd(rm), .z(v00), .status(mul_s1));
    CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
      m01( .a(v0), .b(v2), .rnd(rm), .z(v01), .status(mul_s2));
    CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
      m11( .a(v1), .b(v1), .rnd(rm), .z(v11), .status(mul_s3));

    CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
      a1(.a(v00), .b(v11), .rnd(rm), .z(s1), .status(a_s1));
    CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
      a2(.a(s1), .b(v01), .rnd(rm), .z(result), .status(a_s2));

    assign ready = 1;

endmodule

module m1_a3
  #( int wsig = 23, wexp = 8, ieee = 1, wf = 1 + wexp + wsig )
   ( output uwire [wf-1:0] result,
     output uwire ready,
     input uwire [wf-1:0] v0, v1, v2,
     input uwire start, clk);

    // cadence translate_off
    localparam string name = "One Comb";
    // cadence translate_on
```

```
    localparam int nstages = 1;
    localparam logic [2:0] rm = 0; // Rounding Mode

    uwire [7:0] mul_s1, mul_s2, mul_s3, a_s1, a_s2;
    uwire [wf-1:0] v00, v01, v11, s1;

    CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
      a1(.a(v0), .b(v1), .rnd(rm), .z(s1), .status(a_s1));
    CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
      a2(.a(s1), .b(v2), .rnd(rm), .z(result), .status(a_s2));

    assign ready = 1;

endmodule

module m1_mad
  #( int wsig = 23, wexp = 8, ieee = 1, wf = 1 + wexp + wsig )
   ( output uwire [wf-1:0] result,
     output uwire ready,
     input uwire [wf-1:0] v0, v1, v2,
     input uwire start, clk);

    // cadence translate_off
    localparam string name = "One Comb";
    // cadence translate_on

    localparam int nstages = 1;
    localparam logic [2:0] rm = 0; // Rounding Mode

    uwire [7:0] mul_s1, mul_s2, mul_s3, a_s1, a_s2;
    uwire [wf-1:0] v00, v01, v11, s1;

    CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
      a1(.a(v0), .b(v1), .rnd(rm), .z(s1), .status(a_s1));
    CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
      a2(.a(s1), .b(v2), .rnd(rm), .z(result), .status(a_s2));

    assign ready = 1;

endmodule

module m1_mm
  #( int wsig = 23, wexp = 8, ieee = 1, wf = 1 + wexp + wsig )
   ( output uwire [wf-1:0] p1, p2,
     output uwire ready,
     input uwire [wf-1:0] v0, v1, v2,
     input uwire start, clk);

    // cadence translate_off
    localparam string name = "One Comb";
    // cadence translate_on

    localparam int nstages = 1;
    localparam logic [2:0] rm = 0; // Rounding Mode

    uwire [7:0] mul_s1, mul_s2, mul_s3, a_s1, a_s2;
    uwire [wf-1:0] v00, v01, v11, s1;

    CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
```

```
    a1(.a(v0), .b(v1), .rnd(rm), .z(p1), .status(a_s1));
  CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
    a2(.a(v0), .b(v2), .rnd(rm), .z(p2), .status(a_s2));

  assign ready = 1;

endmodule

module m1_comb
  #( int wsig = 23, wexp = 8, ieee = 1, wf = 1 + wexp + wsig )
   ( output uwire [wf-1:0] result,
     output uwire ready,
     input uwire [wf-1:0] v0, v1,
     input uwire start, clk);

  // cadence translate_off
  localparam string name = "MM Comb";
  // cadence translate_on

  localparam int nstages = 1;
  localparam logic [2:0] rm = 0; // Rounding Mode
  uwire [wf-1:0]  v01, p1, p2;
  uwire [7:0] mul_s2, a_s3;

  CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
    m01( .a(v0), .b(v1), .rnd(rm), .z(v01), .status(mul_s2));

  m1_comb_ssp #( .wsig(wsig), .wexp(wexp), .ieee(ieee) )
  a3( result, ready, v0, v1, v0, start, clk );

  //  assign ready = 1;

endmodule

module m1_v3
  #( int wsig = 23, wexp = 8, ieee = 1, wf = 1 + wexp + wsig )
   ( output uwire [wf-1:0] result,
     output uwire ready,
     input uwire [wf-1:0] v0, v1,
     input uwire start, clk);

  // cadence translate_off
  localparam string name = "MM Comb";
  // cadence translate_on

  localparam int nstages = 1;
  localparam logic [2:0] rm = 0; // Rounding Mode
  uwire [wf-1:0]  v01, p1, p2;
  uwire [7:0] mul_s2, a_s3;

  CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
    m01( .a(v0), .b(v1), .rnd(rm), .z(v01), .status(mul_s2));

  m1_comb_v3 #( .wsig(wsig), .wexp(wexp), .ieee(ieee) )
  a3( result, ready, v0, v1, v01, start, clk );

  //  assign ready = 1;

endmodule
```

```
module m1_comb_mm
  #( int wsig = 23, wexp = 8, ieee = 1, wf = 1 + wexp + wsig )
   ( output uwire [wf-1:0] result,
     output uwire ready,
     input uwire [wf-1:0] v0, v1,
     input uwire start, clk);

   // cadence translate_off
   localparam string name = "MM Comb";
   // cadence translate_on

   localparam int nstages = 1;
   localparam logic [2:0] rm = 0; // Rounding Mode
   uwire [wf-1:0]  v01, p1, p2;
   uwire [7:0] mul_s2, a_s3;

   CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     m01( .a(v0), .b(v1), .rnd(rm), .z(v01), .status(mul_s2));

   m1_mm #( .wsig(wsig), .wexp(wexp), .ieee(ieee) )
   a3( p1, p2, ready, v0, v1, v01, start, clk );

   CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
   a2(.a(p1), .b(p2), .rnd(rm), .z(result), .status(a_s3));


   //  assign ready = 1;

endmodule

module m1_comb_a3
  #( int wsig = 23, wexp = 8, ieee = 1, wf = 1 + wexp + wsig )
   ( output uwire [wf-1:0] result,
     output uwire ready,
     input uwire [wf-1:0] v0, v1,
     input uwire start, clk);

   // cadence translate_off
   localparam string name = "A3 Comb";
   // cadence translate_on

   localparam int nstages = 1;
   localparam logic [2:0] rm = 0; // Rounding Mode
   uwire [wf-1:0]  v01;
   uwire [7:0] mul_s2;

   CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     m01( .a(v0), .b(v1), .rnd(rm), .z(v01), .status(mul_s2));

   m1_a3 #( .wsig(wsig), .wexp(wexp), .ieee(ieee) )
   a3( result, ready, v0, v1, v01, start, clk );

   //  assign ready = 1;

endmodule

module m1_comb_orig
  #( int wsig = 23, wexp = 8, ieee = 1, wf = 1 + wexp + wsig )
```

```
    ( output uwire [wf-1:0] result,
      output uwire ready,
      input uwire [wf-1:0] v0, v1,
      input uwire start, clk);

    // cadence translate_off
    localparam string name = "One Comb";
    // cadence translate_on

    localparam int nstages = 1;
    localparam logic [2:0] rm = 0; // Rounding Mode

    uwire [7:0] mul_s1, mul_s2, mul_s3, a_s1, a_s2;
    uwire [wf-1:0] v00, v01, v11, s1;

    CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
      m01( .a(v0), .b(v1), .rnd(rm), .z(v01), .status(mul_s2));
    CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
      m11( .a(v1), .b(v1), .rnd(rm), .z(v11), .status(mul_s3));

    CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
      a1(.a(v0), .b(v11), .rnd(rm), .z(s1), .status(a_s1));
    CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
      a2(.a(s1), .b(v01), .rnd(rm), .z(result), .status(a_s2));

    assign ready = 1;

endmodule


module m1_seq
  #( int wsig = 23, wexp = 8, ieee = 1, wf = 1 + wexp + wsig )
   ( output logic [wf-1:0] result,  output logic ready,
     input uwire [wf-1:0] v0, v1,   input uwire start, clk );

    // cadence translate_off
    localparam string name = "One Seq";
    // cadence translate_on

    uwire [7:0] mul_s, add_s; // Operation status. Ignored.
    uwire [wf-1:0] mul_a, mul_b, add_a, add_b, prod, sum;

    logic [2:0]  step;
    logic [wf-1:0] ac0, ac1;

    localparam int last_step = 4;

    always_ff @( posedge clk )
      if ( start ) step <= 0;
      else if ( step < last_step ) step <= step + 1;

    localparam logic [2:0] rm = 0; // Rounding Mode
    CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
      m1( .z(prod), .a(mul_a), .b(mul_b), .rnd(rm), .status(mul_s));
    CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
      a1( .z(sum),  .a(add_a), .b(add_b), .rnd(rm), .status(add_s));

    localparam logic [wf-1:0] one = { ( 1 << wexp - 1 ) - 1, wsig'(0) };
    assign mul_a = step < 2  ? v0 : v1;
```

```
   assign mul_b = step == 0 ? one : v1;
   assign add_a = ac0,  add_b = ac1;

   always_ff @( posedge clk )
     begin
        ac0 <= prod;
        if ( step < 3 ) ac1 <= step ? sum : 0;
        if ( start ) ready <= 0; else if ( step == last_step-1 ) ready <= 1;
     end

   assign result = sum;

endmodule

module m1_pipe
  #( int wsig = 23, wexp = 8, ieee = 1, wf = 1 + wexp + wsig )
   ( output uwire [wf-1:0] result,
     output uwire ready,
     input uwire [wf-1:0] v0, v1,
     input uwire start, clk);

   // cadence translate_off
   localparam string name = "One Pipe";
   // cadence translate_on

   localparam int nstages = 4;
   localparam logic [2:0] rm = 0; // Rounding Mode

   uwire [7:0] mul_s1, mul_s2, mul_s3, a_s1, a_s2;
   uwire [wf-1:0] v00, v01, v11, s1, s2;
   logic [wf-1:0] pl_1_v00, pl_1_v01, pl_1_v11;
   logic [wf-1:0] pl_2_v0001, pl_2_v11;
   logic [wf-1:0] pl_3_sum;
   logic pl_1_occ, pl_2_occ, pl_3_occ;

   CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     m01( .z(v01), .a(v0), .b(v1), .rnd(rm), .status(mul_s2));
   CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     m11( .z(v11), .a(v1), .b(v1), .rnd(rm), .status(mul_s3));

   CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     a1(.z(s1), .a(pl_1_v00),   .b(pl_1_v01), .rnd(rm), .status(a_s1));
   CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     a2(.z(s2), .a(pl_2_v0001), .b(pl_2_v11), .rnd(rm), .status(a_s2));

   assign ready = pl_3_occ;
   assign result = pl_3_sum;

   always_ff @( posedge clk ) begin

      pl_1_v00 <= v0;
      pl_1_v01 <= v01;
      pl_1_v11 <= v11;
      pl_1_occ <= start;

      pl_2_v0001 <= s1;
      pl_2_v11 <= pl_1_v11;
      pl_2_occ <= pl_1_occ;
```

```
      pl_3_sum <= s2;
      pl_3_occ <= pl_2_occ;

   end

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Testbench Code
//
// It is okay to modify the testbench code to facilitate the coding
// and debugging of your modules.


// cadence translate_off

function automatic real rand_real(real minv, real maxv);
      rand_real = minv + ( maxv - minv ) * ( real'({$random}) ) / 2.0**32;
endfunction

function automatic real fabs(real val);
      fabs = val < 0 ? -val : val;
endfunction

virtual class conv #(int wexp=6, wsig=10);
   // Convert between real and fp types using parameter-provided
   // exponent and significand sizes.

   localparam int w = 1 + wexp + wsig;
   localparam int bias_r = ( 1 << 11 - 1 ) - 1;
   localparam int w_sig_r = 52;
   localparam int w_exp_r = 11;
   localparam int bias_h = ( 1 << wexp - 1 ) - 1;

   static function logic [w-1:0] rtof( real r );
      logic [wsig-1:0] sig_f;
      logic [w_sig_r-wsig-1:0] sig_x;
      logic [w_exp_r-1:0] exp_r;
      logic sign_r;
      { sign_r, exp_r, sig_f, sig_x } = $realtobits(r);
      rtof = !r ? 0 : { sign_r, wexp'( exp_r + bias_h - bias_r ), sig_f };
   endfunction

   static function real ftor( logic [w-1:0] f );
      ftor = !f ? 0.0
        : $bitstoreal
          ( { f[w-1],
              w_exp_r'( bias_r + f[w-2:wsig] - bias_h ),
              f[wsig-1:0], (w_sig_r-wsig)'(0) } );
   endfunction

endclass


program reactivate
   (output uwire clk_reactive, output int cycle_reactive,
    input uwire clk, input var int cycle);
   assign clk_reactive = clk;
```

```
   assign cycle_reactive = cycle;
endprogram

module testbench;

   localparam int npsets = 4; // Number of instantiations.
   localparam int pset[npsets][2] =
              '{ { 7, 0 }, { 23, 0}, {7, 1 }, {23, 1} };
   //
   // Above: First number in each pair is value of n_avg_of,
   // second number is maximum word length.

   int n_err_shown;  // Number of times error info printed to console.
   int n_err_sh_nc, n_err_sh_nw, n_err_sh_avg, n_err_sh_state;
   initial begin
      n_err_sh_nc = 0;
      n_err_sh_nw = 0;
      n_err_sh_avg = 0;
      n_err_sh_state = 0;
   end
   int t_errs;       // Total number of errors.
   initial begin t_errs = 0; n_err_shown = 0; end
   final $write("Total number of errors: %0d\n",t_errs);

   uwire d[npsets:-1];    // Start / Done signals.
   assign d[-1] = 1;  // Initialize first at true.

   // Instantiate a testbench at each size.
   //
   for ( genvar i=0; i<npsets; i++ )
     testbench_n #(pset[i][0],pset[i][1]) t2( .done(d[i]), .tstart(d[i-1]) );

endmodule



module testbench_n
  #( int w_sig = 7, use_one = 0 )
   ( output logic done, input uwire tstart );

   typedef enum { MT_comb, MT_seq, MT_pipe } Module_Type;

   localparam int w_exp = 8;
   localparam int wid = w_sig + w_exp + 1;
   localparam int max_latency = 10;
   localparam int num_tests = 16;
   localparam int nmuts = 10;
   int err[nmuts];

   uwire [wid-1:0]  mag[nmuts];
   uwire         ready[nmuts];
   real   magr;
   real vr[2];
   logic [wid-1:0] v[2], vp[2];
   logic        start;

   typedef struct
     {
      int idx;
```

```
   int err_count = 0;
   int ncyc = 0;
   Module_Type mt = MT_comb;
   logic [wid-1:0] sout = 'h111;
   int cyc_tot = 0;
   } Info;
Info pi[string];

localparam int cycle_limit = num_tests * max_latency * 4;
int cycle, cyc_start;
logic clock;
bit use_others;

logic clk_reactive;
int cycle_reactive;
reactivate ra(clk_reactive,cycle_reactive,clock,cycle);

task pi_seq(input int idx, input string name);
   automatic string m = $sformatf("%s", name);
   pi[m].idx = idx; pi[m].mt = MT_seq;
endtask

task pi_comb(input int idx, input string name);
   automatic string m = $sformatf("%s", name);
   pi[m].idx = idx; pi[m].mt = MT_comb;
endtask

task pi_pipe(input int idx, input string name, input int ncyc);
   automatic string m = $sformatf("%s", name);
   pi[m].idx = idx; pi[m].mt = MT_pipe;
   pi[m].ncyc = ncyc;
endtask

initial begin
   clock = 0;
   cycle = 0;

   done = 0;
   wait( tstart );

   fork
      while ( !done ) #10 cycle += clock++;
      wait( done );
      wait( cycle >= cycle_limit )
        $write("*** Cycle limit exceeded, ending.\n");
   join_any;

   done = 1;
end

if ( use_one ) begin

   m1_functional mf( magr, vr[0], vr[1] );
   m1_seq #( .wsig(w_sig), .wexp(w_exp), .ieee(0) )
     m2( mag[1], ready[1], v[0],v[1], start, clock );
   initial begin pi_seq(1,m2.name); end

   m1_comb_alt #( .wsig(w_sig), .wexp(w_exp), .ieee(0) )
     m5r( mag[5], ready[5], v[0],v[1], start, clock );
```

```
      initial begin pi_comb(5,m5r.name); end

      m1_pipe #( .wsig(w_sig), .wexp(w_exp), .ieee(0) )
        m3( mag[3], ready[3], vp[0],vp[1], start, clock );
      initial begin pi_pipe(3,m3.name,m3.nstages); end

   end else begin

      ms_functional mf( magr, vr[0], vr[1] );
      ms_seq #( .wsig(w_sig), .wexp(w_exp), .ieee(0) )
        m2( mag[1], ready[1], v[0],v[1], start, clock );
      initial begin pi_seq(1,m2.name); end

      ms_comb #( .wsig(w_sig), .wexp(w_exp), .ieee(0) )
        m5r( mag[5], ready[5], v[0],v[1], start, clock );
      initial begin pi_comb(5,m5r.name); end

      ms_pipe #( .wsig(w_sig), .wexp(w_exp), .ieee(0) )
        m3( mag[3], ready[3], vp[0],vp[1], start, clock );
      initial begin pi_pipe(3,m3.name,m3.nstages); end

   end

   initial begin

      while ( !done ) @( posedge clk_reactive ) #2

         if ( use_others ) begin

            vp = v;
            use_others = 0;
            start = 1;

         end else begin

            vp[0] = conv#(w_exp,w_sig)::rtof( real'(cycle-cyc_start) );
            vp[1] = cycle - cyc_start;
            start = 0;

         end
   end

   initial begin

      automatic int n_err = 0;

      use_others = 0;
      start = 0;

      @( posedge clk_reactive );


      for ( int i=0; i<num_tests; i++ ) begin

         automatic int awaiting = pi.num();

         cyc_start = cycle;

         if ( i < 4 ) begin
```

```
      // In first eight test vector components are zero or one.
      //
      for ( int j=0; j<2; j++ ) vr[j] = i & 1 << j ? 1.0 : 0.0;

   end else begin

      // In other tests vector components are randomly chosen.
      //
      for ( int j=0; j<2; j++ ) vr[j] = rand_real(-10,+10);

   end

   for ( int j=0; j<2; j++ ) v[j] = conv#(w_exp,w_sig)::rtof(vr[j]);

   vp = v;
   use_others = 1;

   /// Collect Result (mag) From Each Module Under Test (mut)
   //
   foreach ( pi[muti] ) begin

      // Note: need to make a local copy of muti because of the
      // fork below.
      automatic string mut = muti;

      // Create a child thread to get response from current mut.
      // The parent thread, without delay, proceeds to join_none.
      //
      fork begin

         if ( pi[mut].mt == MT_seq ) begin

            wait ( !ready[pi[mut].idx] );
            wait ( ready[pi[mut].idx] );

         end else begin

            // Compute time at which result should be ready or
            // when to start examining a READY output.
            //
            automatic int latency =
              pi[mut].mt == MT_comb ? 1 : pi[mut].ncyc;
            automatic int eta = cyc_start + latency;

            wait ( cycle_reactive == eta );

         end

         // Decrement count of the number of modules we are waiting for.
         //
         awaiting--;

         // Store the module MAG output, it will be checked later
         // for correctness.
         //
         pi[mut].sout = mag[pi[mut].idx];

         pi[mut].cyc_tot += cycle - cyc_start;
```

```
            // This thread ends execution here.
        end join_none;

    end

    // Wait until data collected from all modules under test.
    //
    wait ( awaiting == 0 );

    // Check the output of each Module Under Test.
    //
    foreach ( pi[ mut ] ) begin

       // Assign module output to a real.
       //
       automatic real mmagr = conv#(w_exp,w_sig)::ftor(pi[mut].sout);
       //
       // Note: pi[mut].sout is type logic which is assumed to be
       // an unsigned integer. However, the contents is really an
       // float.

       // Compute difference between module output and expected
       // output.  With FP small differences can be okay, they might
       // occur, for example, due to differences in the order of
       // operations.
       //
       automatic real err_mag =
         fabs( mmagr - magr ) / fabs( magr ? magr : 1 );
       localparam real tol = real'(4) / ( 1 << w_sig );
       automatic bit okay = err_mag < tol;

       if ( !okay ) begin
         pi[mut].err_count++;
          n_err++;
         if ( pi[mut].err_count < 5 )
           $write("%s (%0d) test #%0d vec (%.1f,%.1f) error: h'%8h  %7.4f != %7.4f (correct)\n",
              mut, w_sig, i, vr[1], vr[0],
                  pi[mut].sout, mmagr, magr);
      end
    end

    while ( {$random} & 1 == 1 ) @( posedge clk_reactive );
    //
    // Note: By waiting for reactive clock we can be sure that
    // modules under test have completed all work due to the
    // positive edge of the regular clk. Wait a random amount of
    // time in case any modules are only correct at some stride.

end

foreach ( pi[ mut ] )
  $write("Ran %4d tests for (%0d) %-0s, %4d errors found. Avg cyc %.1f\n",
         num_tests, w_sig, mut,
         pi[mut].err_count,
         pi[mut].mt == MT_comb ? 1 : real'(pi[mut].cyc_tot) / num_tests);

done = 1;
testbench.t_errs += n_err;
```

```
    end

endmodule

`define SIMULATION_ON

// cadence translate_on

`default_nettype wire

`ifdef SIMULATION_ON

`include "/apps/linux/cadence/GENUS211/share/synth/lib/chipware/sim/verilog/CW/CW_fp_mult.v"
`include "/apps/linux/cadence/GENUS211/share/synth/lib/chipware/sim/verilog/CW/CW_fp_add.v"

`else

`include "/apps/linux/cadence/GENUS211/share/synth/lib/chipware/syn/CW/CW_fp_mult.v"
`include "/apps/linux/cadence/GENUS211/share/synth/lib/chipware/syn/CW/CW_fp_add.v"

`endif
```

17 Fall 2021 Solutions

LSU EE 4755 Homework 1 Solution Due: 24 September 2021

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/2021/hw01.v.html`.

Problem 0: Following instructions at `https://www.ece.lsu.edu/koppel/v/proc.html`, set up your class account, copy the assignment, and run the Verilog simulator and synthesis program on the unmodified homework file, `hw01.v`. Do this early enough so that minor problems (*e.g.*, password doesn't work) are minor problems.

Problem 1: The partially completed `insert_at` module below and in the homework assignment file has three inputs, a `wa`-bit input `ia`, a `wb`-bit input `ib`, and a $\lceil \lg(\texttt{wa}+1) \rceil$-bit input `pos`, and there is one output, a `wa+wb`-bit output `o`. Complete the module following the coding requirements given further below so that `o` consists of the bits of `ia` with `ib` inserted at `pos`. That is, `o[pos-1:0]` should be set to `ia[pos-1:0]`, `o[wb+pos-1:pos]` should be set to `ib`, and `o[wa+wb-1:wb+pos]` should be set to `ia[wa-1:pos]`.

For example, let `wa=6` and `wb=2`, `ia=111111`, `ib=00`, and `pos = 2`. Then `o=11110011`. For `pos=5`, `o=10011111`. For those still not 100% sure of what `o` should be set to should look at how `o_shadow` is computed in the `testbench` module. Also, the testbench will show what the output should be when it isn't.

```
module insert_at
  #( int wa = 20, wb = 10, wo = wa+wb, walg = $clog2(wa+1) )
   ( output logic [wo-1:0] o,
     input uwire [wa-1:0] ia, input uwire [wb-1:0] ib,
     input uwire [walg-1:0] pos );

   // The line assigning mask_low must be replaced with a mask module.
   uwire [wo-1:0] mask_low = ( 1 << pos ) - 1;  // REPLACE ME!

   uwire [wo-1:0] ib_at_pos;
   shift_left #(wb,wo,walg) sl1( ib_at_pos, ib, pos );

   assign o = ia & mask_low | ib_at_pos;
endmodule
```

The `insert_at` module must be synthesizable and must not use procedural code and must not use shift operators. (That includes the line assigning `mask_low`, it must be replaced.) Instead, rely on instantiations of the provided shift and mask modules.

The testbench will test your module and report the first few errors. For example, here is the testbench output for the unmodified module:

```
Error for ia=11111111  ib=000  pos= 0  00000000000 != 11111111000 (correct)
Error for ia=11111111  ib=000  pos= 1  00000000001 != 11111110001 (correct)
Error for ia=11111111  ib=000  pos= 2  00000000011 != 11111100011 (correct)
Error for ia=11111111  ib=000  pos= 3  00000000111 != 11111000111 (correct)
Error for ia=11111111  ib=000  pos= 4  00000001111 != 11110001111 (correct)
Done with 27 tests, 15 errors found.
```

The text `00000001111 != 11110001111 (correct)` shows the output of `insert_at` to the left of the `!=` and the correct answer to the right. So in this case `00000001111` is the module output

and 11110001111 is what the module output should have been. Only the first few errors are shown, but the total number of errors is reported at the end, 15 in this case.

Synthesizability can be checked by running the synthesis script using the command `genus -files syn.tcl`. If the module is synthesizable (though not necessarily correct) a table of area and delay will be shown, for example:

```
Module Name                              Area   Delay   Delay
                                              Actual  Target
insert_at                               51832   0.987   1.000 ns
insert_at_1                             97968   0.616   0.100 ns
Normal exit.
```

One common problem encountered by beginners is setting the correct port sizes. For example, the `shift_left` module the port sizes are `wi`, `wo`, and `wolg`:

```
module insert_at #( int wa = 20, wb = 10, wo = wa+wb, walg = $clog2(wa+1) )
   ( output logic [wo-1:0] o,
     input uwire [wa-1:0] ia, input uwire [wb-1:0] ib,
     input uwire [walg-1:0] pos );
   uwire [wo-1:0] ib_at_pos;
   shift_left #(wb,wo,walg) sl1( ib_at_pos, ib, pos );
```

So the first connection to a `shift_left` instantiation must be `wi` bits, the second must be `wo` bits, and the third `wolg` bits. In the unmodified `insert_at` these parameters to `insert_at` were set explicitly to match the connection sizes. Sometimes it may be necessary to use an intermediate object or to cast in order to get the correct connection size. For example, if we wanted to shift by `pos+1` the following would not work:

```
   shift_left #(wb,wo,walg) sl1( ib_at_pos, ib, pos + 1 );
```

because the `1` in the `pos+1` expression implicitly expands it to 32 bits. (This results in a warning, but it's not good to clutter compiler output with ignorable warnings.) The problem can be solved using a cast:

```
   shift_left #(wb,wo,walg) sl1( ib_at_pos, ib, walg'(pos + 1) );
```

Solution starts on the next page.

The solution appears below, and can be found in the assignment directory, and on the course Web pages at `https://www.ece.lsu.edu/koppel/v/2021/hw01-sol.v.html`. Immediately below is the solution without extensive comments. On the following pages is the same solution, but with sample values shown in the comments.

```
module insert_at
  #( int wa = 20, wb = 10, wo = wa+wb, walg = $clog2(wa+1) )
   ( output logic [wo-1:0] o,
     input uwire [wa-1:0] ia,
     input uwire [wb-1:0] ib,
     input uwire [walg-1:0] pos );
   ///                          SOLUTION
   uwire [wa-1:0] mask_low;
   mask_lsb #(wa) ml(mask_low, pos);
   uwire [wa-1:0] ia_low = ia & mask_low;
   uwire [wa-1:0] ia_high_low = ia & ~mask_low;

   localparam int wblg = $clog2(wb);
   uwire [wo-1:0] ia_high;
   shift_left #(wa,wo,wblg) slc( ia_high, ia_high_low, wblg'(wb) );

   uwire [wo-1:0] ib_at_pos;
   shift_left #(wb,wo,walg) slb( ib_at_pos, ib, pos );

   assign o = ia_high | ib_at_pos | ia_low;
endmodule
```

The challenge in this assignment was refreshing your knowledge of Verilog and digital logic. If you can't follow the module above, look at the one on the following pages and in particular use the sample values to figure out what is going one.

The solution here makes use of a single mask unit (named `ml`) creating mask `mask_low`. This mask is used twice, in its original form to extract the lowest `pos` bits of `ia` into `ia_low` and in inverted form to extract the high bits of `ia` into `ia_high_low`. Note that both `ia_low` and `ia_high_low` are `wa`-bit quantities. The "shifter" `slc` writes a shifted value of `ia_high_low` into `ia_high`. Notice that the shift-amount input to `slc` (the last port) is `wb`, a constant (since it's a module parameter). That brings the cost of `slc` to zero.

A real shifter, `slb`, is used to move `ib` into the correct position in its output `ib_at_pos`. The assign statement puts all of these together.

Common Mistakes: In a few solutions the shift amounts or mask sizes were set assuming that `wa=8` and `wb=3`. That is not correct because `insert_at` can be instantiated with other possible values of `wa` and `wb`.

Another common mistake was to set the width of the shift amount port to a value much larger than needed. For example, consider:

```
shift_left #(wb,wa+wb,wo) slb( ib_at_pos, ib, phat_pos );
```

The third parameter of the `shift_left` module has been set to `wo`, which is overkill. (The shift amount input has been renamed `phat_pos` to emphasize its new size.) For this use of `shift_left` the most by which we would shift is `ia` bits, so at most the position would take $\lceil \log\_2 \mathrm{wa} \rceil$ (or as a Verilog expression, `$clog2(wa)`) bits. Setting a parameter like this to too large a value will not affect correctness (in cases like this) but it can increase the cost of the synthesized hardware. That depends on the synthesis programs ability to recognize that high-order bits will always be zero. So for that reason it is best to set parameters to appropriate values. That does mean taking the time to learn what each parameter is for and to set it properly, but that is what you would be paid for.

Solution with sample values appearing in the comments:

```
module insert_at
  #( int wa = 20, wb = 10, wo = wa+wb, walg = $clog2(wa+1) )
   ( output logic [wo-1:0] o,
     input uwire [wa-1:0] ia,
     input uwire [wb-1:0] ib,
     input uwire [walg-1:0] pos );

   /// SOLUTION
   /// :Example: Input Values:
   //
   //  ia =              aaaaaaaa   (Each a is a bit of ia, it can be 0 or 1 .)
   //  ib =                   bbb   (Each b is a bit of ib, it can be 0 or 1 .)
   //  pos = 2
   //
   /// Desired Output Value
   //
   //  o      =       aaaaaabbbaa   (Notice that ib is insert at pos 2)

   uwire [wa-1:0] mask_low;
   mask_lsb #(wa) ml(mask_low, pos);
   uwire [wa-1:0] ia_low = ia & mask_low;
   uwire [wa-1:0] ia_high_low = ia & ~mask_low;

   //  ia =              aaaaaaaa
   //  mask_low =        00000011  (Two low bits are 1 because pos=2.)
   //  ia_low =          000000aa  (ia_low has the bits to the right of pos.)
   //  ia_high_low =     aaaaaa00  (ia_high_low: the bits to the left of pos.)

   localparam int wblg = $clog2(wb);
   uwire [wo-1:0] ia_high;
   shift_left #(wa,wo,wblg) slc( ia_high, ia_high_low, wblg'(wb) );

   //  ia_high_low =     aaaaaa00
   //  ia_high =      aaaaaa00000 (Shift wb bits to make room for ib.)

   uwire [wo-1:0] ib_at_pos;
   shift_left #(wb,wo,walg) slb( ib_at_pos, ib, pos );

   //  ib =                   bbb
   //  ib_at_pos =    000000bbb00 (Shifted pos bits, and widened to wo bits.)

   assign o = ia_high | ib_at_pos | ia_low;

   //  ia_high =      aaaaaa00000
   //  ib_at_pos =    000000bbb00
   //  ia_low =          000000aa
   //  o      =       aaaaaabbbaa
endmodule
```

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2021 Homework 1
/// SOLUTION

 /// Assignment https://www.ece.lsu.edu/koppel/v/2021/hw01.pdf
 /// Solution Discussion https://www.ece.lsu.edu/koppel/v/2021/hw01_sol.pdf

`default_nettype none

////////////////////////////////////////////////////////////////////////////////
/// Problem 1
//
 ///  Complete insert_at so that output o is set to ia with ib inserted at pos.
 ///
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//
//     [✔] Do not use procedural code.
//     [✔] Do not use the << or >> operators (or anything similar).
//     [✔] Use the shift and mask modules to provide shifted values
//         and bitmasks.
//
//     [✔] Don't assume any particular parameter value.
//
//     [✔] Code must be written clearly.
//     [✔] Pay attention to cost and performance.


module insert_at
  #( int wa = 20, wb = 10, wo = wa+wb, walg = $clog2(wa+1) )
   ( output logic [wo-1:0] o,
     input uwire [wa-1:0] ia,
     input uwire [wb-1:0] ib,
     input uwire [walg-1:0] pos );

   /// SOLUTION

   /// :Example: Input Values:
   //
   //  ia =              aaaaaaaa   (Each a is a bit of ia, it can be 0 or 1.)
   //  ib =                   bbb   (Each b is a bit of ib, it can be 0 or 1.)
   //  pos = 2
   //
   /// Desired Output Value
   //
   //  o      =       aaaaaabbbaa   (Notice that ib is inserted at pos 2.)

   uwire [wa-1:0] mask_low;
   mask_lsb #(wa) ml(mask_low, pos);
```

```
   uwire [wa-1:0] ia_low = ia & mask_low;
   uwire [wa-1:0] ia_high_low = ia & ~mask_low;

   //  ia =              aaaaaaaa
   //  mask_low =        00000011  (Two low bits are 1 because pos=2.)
   //  ia_low =          000000aa  (ia_low has the bits to the right of pos.)
   //  ia_high_low =     aaaaaa00  (ia_high_low: the bits to the left of pos.)

   localparam int wblg = $clog2(wb);
   uwire [wo-1:0] ia_high;
   shift_left #(wa,wo,wblg) slc( ia_high, ia_high_low, wblg'(wb) );

   //  ia_high_low =     aaaaaa00
   //  ia_high =      aaaaaa00000 (Shift wb bits to make room for ib.)

   uwire [wo-1:0] ib_at_pos;
   shift_left #(wb,wo,walg) slb( ib_at_pos, ib, pos );

   //  ib =                  bbb
   //  ib_at_pos =    000000bbb00 (Shifted pos bits, and widened to wo bits.)

   assign o = ia_high | ib_at_pos | ia_low;

   //  ia_high =      aaaaaa00000
   //  ib_at_pos =    000000bbb00
   //  ia_low =          000000aa
   //  o      =       aaaaaabbbaa

endmodule


module shift_left
  #( int wi = 4, wo = wi, wolg = $clog2(wo) )
   ( output uwire [wo-1:0] o,
     input uwire [wi-1:0] i,
     input uwire [wolg-1:0] amt );
   assign o = i << amt;
endmodule

module shift_right
  #( int wi = 4, wo = wi, wolg = $clog2(wo) )
   ( output uwire [wo-1:0] o,
     input uwire [wi-1:0] i,
     input uwire [wolg-1:0] amt );
   assign o = i >> amt;
endmodule

module mask_lsb
  #( int wo = 6, wp = $clog2(wo+1) )
   ( output logic [wo-1:0] o, input uwire [wp-1:0] n1 );
   always_comb for ( int i=0; i<wo; i++ ) o[i] = i < n1;
endmodule

module mask_msb
```

```
  #( int wo = 6, wp = $clog2(wo+1) )
   ( output logic [wo-1:0] o, input uwire [wp-1:0] n1 );
   always_comb for ( int i=0; i<wo; i++ ) o[wo-i-1] = i < n1;
endmodule



////////////////////////////////////////////////////////////////////////////////
/// Testbench Code

// cadence translate_off

module testbench;

   logic done [1:-1];
   initial done[-1] = 1;

   testbench_size #(8,3, "Set 1") tb1(done[0],done[-1]);
   testbench_size #(4,5, "Set 2") tb2(done[1],done[0]);

endmodule

module testbench_size
  #( int wa = 8, int wb = 3, string label = "set me" )
   ( output logic done_me,
     input uwire logic done_pred );

   localparam int wo = wa+wb;
   localparam int walg = $clog2(wa+1);

   localparam int n_tests = (wa+1) * 3;


   logic [wa-1:0] ia;
   logic [wb-1:0] ib;
   uwire [wo-1:0] o;
   logic [walg-1:0] pos;

   insert_at #(wa,wb) iat(o, ia, ib, pos);

   initial begin

      automatic int n_err = 0;

      wait ( done_pred === 1 );

      for ( int tn = 0; tn < n_tests; tn++ ) begin

         automatic int rnd = tn / (wa+1);

         logic [wo-1:0] o_shadow;

         pos = tn % (wa+1);
         case ( rnd )
```

```
            0: begin ia = -1; ib = 0; end
            1: begin ia = 0;  ib = -1; end
            default: {ia,ib} = {$random};
         endcase

         #1;

         for ( int i=0; i<pos; i++ ) o_shadow[i] = ia[i];
         for ( int i=0; i<wb; i++ ) o_shadow[i+pos] = ib[i];
         for ( int i=pos; i<wa; i++ ) o_shadow[i+wb] = ia[i];

         if ( o_shadow !== o ) begin
            n_err++;
            if ( n_err < 6 )
              $write("Error %s for ia=%b  ib=%b  pos=%d  %b != %b (correct)\n",
                     label,
                     ia, ib, pos, o, o_shadow);
         end

      end

      $write("For %s, done with %0d tests, %0d errors found.\n",
             label, n_tests, n_err);

      done_me = 1;

   end

endmodule

// cadence translate_on
```

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2021 Homework 2
//
 /// SOLUTION

 /// Assignment https://www.ece.lsu.edu/koppel/v/2021/hw02.pdf

`default_nettype none


////////////////////////////////////////////////////////////////////////////////
/// Problem 1
//
 ///  Complete nn_sparse so that it computes both dense (fmt=4'b1111)
 ///  and sparse (fmt= 4'b1100, 4'b0110, 4'b1010, etc.) products.
 ///
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//
//     [✔] To achieve the fastest speed a sparse product computation
//         should not go through two adders.
//
//     [✔] Don't assume any particular parameter value.
//
//     [✔] Code must be written clearly.
//     [✔] Pay attention to cost and performance.


module nn_sparse
  #( int nn = 4, wexp = 6, wsig_ac = 15, wsig_in = 10, wsig_wd = 6,
     wo = 1 + wexp + wsig_ac,
     wi = 1 + wexp + wsig_in,
     ww = nn * ( 1 + wexp + wsig_wd ) )
   ( output logic [wo-1:0] o,
     input uwire [wi-1:0] i[nn],
     input uwire [ww-1:0] w,
     input uwire [nn-1:0] fmt );

   // Compute size of significand of sparse weights.
   localparam int wsig_ws = 2 * wsig_wd + wexp + 1;

   // Separate w into dense weights.
   //
   localparam int wwd = ww / nn;
   uwire [3:0][wwd-1:0] wd;
   assign wd = w;

   // SOLUTION
   //
   // Separate w into sparse weights
   //
```

```
   localparam int wws = wwd * 2;
   uwire [1:0][wws-1:0] ws = w;

   // Dense
   uwire [wo-1:0] acc1, acc2, od, os;
   nn2 #(wexp,wsig_in,wsig_wd,wsig_ac) nn2d1(acc1, i[0], i[1], wd[0], wd[1]);
   nn2 #(wexp,wsig_in,wsig_wd,wsig_ac) nn2d2(acc2, i[2], i[3], wd[2], wd[3]);
   fp_add #(wexp,wsig_ac) add(od,acc1,acc2);

   // SOLUTION
   //
   // Select the two inputs that will participate in the sparse
   // computation ..
   //
   uwire [wi-1:0] is0 = fmt[0] ? i[0] : fmt[1] ? i[1] : i[2];
   uwire [wi-1:0] is1 = fmt[3] ? i[3] : fmt[2] ? i[2] : i[1];
   //
   // .. and connect them to an nn2 instantiation in which the weight
   // input size parameters are wsig_ws instead of wsig_wd.
   //
   nn2 #(wexp,wsig_in,wsig_ws,wsig_ac) nn2s(os, is0, is1, ws[0], ws[1]);

   // SOLUTION
   //
   // Route the appropriate value to the output.
   //
   assign o = fmt[2:0] == 3'b111 ? od : os;

endmodule

module nn_sparse_cheap
  #( int nn = 4, wexp = 6, wsig_ac = 15, wsig_in = 10, wsig_wd = 6,
     wo = 1 + wexp + wsig_ac,
     wi = 1 + wexp + wsig_in,
     ww = nn * ( 1 + wexp + wsig_wd ) )
   ( output logic [wo-1:0] o,
     input uwire [wi-1:0] i[nn],
     input uwire [ww-1:0] w,
     input uwire [nn-1:0] fmt );

   // This module is less expensive than nn_sparse because it
   // instantiates only two nn2 modules, but it has a longer
   // critical path.

   localparam int wwd = ww / nn;

   localparam int wsig_ws = 2 * wsig_wd + wexp + 1;
   localparam int wws = 1 + wexp + wsig_ws;

   uwire sparse = &fmt[2:0] == 0;

   uwire [3:0][wwd-1:0] wd; // Xcelium bug?: can't assign on decl line.
   assign wd = w;
   uwire [1:0][wws-1:0] ws = w;
```

```
   // Dense
   uwire [wo-1:0] acc1, acc2, od, os;

   nn2 #(wexp,wsig_in,wsig_wd,wsig_ac) nn2d2(acc2, i[2], i[3], wd[2], wd[3]);
   fp_add #(wexp,wsig_ac) add(od,acc1,acc2);


   uwire [wi-1:0] is0 = fmt[0] ? i[0] : fmt[1] ? i[1] : i[2];
   uwire [wi-1:0] is1 = !sparse ? i[1] : fmt[3] ? i[3] : fmt[2] ? i[2] : i[1];

   uwire [wws-1:0] ws0 = sparse ? ws[0] : wd[0] << wsig_ws - wsig_wd;
   uwire [wws-1:0] ws1 = sparse ? ws[1] : wd[1] << wsig_ws - wsig_wd;


   // Sparse
   nn2 #(wexp,wsig_in,wsig_ws,wsig_ac) nn2s(acc1, is0, is1, ws0, ws1 );

   assign o = sparse ? acc1 : od;

endmodule



module nn2
  #( int wexp = 9, wsig_in = 10, wsig_w = 5, wsig_ac = 12,
     wi = 1 + wexp + wsig_in,
     ww = 1 + wexp + wsig_w,
     wo = 1 + wexp + wsig_ac)
   ( output uwire [wo-1:0] o,
     input uwire [wi-1:0] i0, i1,
     input uwire [ww-1:0] w0, w1 );

   uwire [wo-1:0] p0, p1;
   hy_mult #(wexp, wsig_in, wsig_w, wsig_ac) m0(p0,i0,w0);
   hy_mult #(wexp, wsig_in, wsig_w, wsig_ac) m1(p1,i1,w1);
   fp_add #(wexp,wsig_ac) a(o,p0,p1);

endmodule

module fp_add
  #( int wexp = 3, wsig = 50, w = 1 + wexp + wsig )
   ( output uwire [w-1:0] sum,
     input uwire [w-1:0] i0, i1 );

   uwire [7:0] s;
   localparam logic [2:0] rnd_to_0 = 3'b1;

   CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(0))
   a(.a(i0),.b (i1), .rnd (rnd_to_0), .z (sum), .status (s) );

endmodule

module hy_mult
```

```
  #( int wexp = 5, int wsig_a = 6, int wsig_b = 7,
     int wsig_p = wsig_a + wsig_b )
   ( output uwire [wexp+wsig_p:0] prod,
     input uwire [wexp+wsig_a:0] a,
     input uwire [wexp+wsig_b:0] b );

   uwire [7:0] s;
   localparam logic [2:0] rnd_to_0 = 3'b1;
   localparam logic [2:0] rnd_to_plus_inf = 3'b10;
   localparam logic [2:0] rnd_to_minus_inf = 3'b11;

   localparam int wm = 1 + wexp + wsig_p;
   localparam int wsig_diff_a = wsig_p - wsig_a;
   localparam int wsig_diff_b = wsig_p - wsig_b;
   uwire [wm-1:0] ea = wsig_diff_a >= 0
                ? a << wsig_diff_a : a[wexp+wsig_a:-wsig_diff_a];
   uwire [wm-1:0] eb = wsig_diff_b >= 0
                ? b << wsig_diff_b : b[wexp+wsig_b:-wsig_diff_b];

   CW_fp_mult #( .sig_width(wsig_p), .exp_width(wexp), .ieee_compliance(0))
   U1(.a(ea),.b (eb), .rnd (rnd_to_0), .z (prod), .status (s) );

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Testbench Code


// cadence translate_off

virtual class conv #(int wexp=6, wsig=10);
   // Convert between real and fp types using parameter-provided
   // exponent and significand sizes.

   localparam int w = 1 + wexp + wsig;
   localparam int bias_r = ( 1 << 11 - 1 ) - 1;
   localparam int w_sig_r = 52;
   localparam int w_exp_r = 11;
   localparam int bias_h = ( 1 << wexp - 1 ) - 1;

   static function logic [w-1:0] rtof( real r );
      logic [wsig-1:0] sig_f;
      logic [w_sig_r-wsig-1:0] sig_x;
      logic [w_exp_r-1:0] exp_r;
      logic sign_r;
      { sign_r, exp_r, sig_f, sig_x } = $realtobits(r);
      rtof = !r ? 0 : { sign_r, wexp'( exp_r + bias_h - bias_r ), sig_f };
   endfunction

   static function real ftor( logic [w-1:0] f );
      ftor = !f ? 0.0
        : $bitstoreal
          ( { f[w-1],
```

```
            w_exp_r'( bias_r + f[w-2:wsig] - bias_h ),
            f[wsig-1:0], (w_sig_r-wsig)'(0) } );
   endfunction

endclass

function real fabs(real a);
      fabs = a < 0 ? -a : a;
endfunction

function int min( int a, b );
      min = a < b ? a : b;
endfunction

function int min3( int a, b, c );
      automatic int ab = a < b ? a : b;
      min3 = ab < c ? ab : c;
endfunction



module testbench_nn_sparse;

   localparam int npsets = 3;
   localparam int pset[npsets][4] =
            '{ {5, 20, 15, 4}, {6, 18, 10, 5}, {6, 18, 12, 3} };
   // wexp, wsig_ac, wsig_in, wsig_wd
   logic done[npsets:0];

   initial done[0] = 1;

   for ( genvar i = 0; i<npsets; i++ )
      testbench_nn_sparse_p
        #(pset[i][0],pset[i][1],pset[i][2],pset[i][3])
        tb(done[i+1],done[i]);


endmodule

module testbench_nn_sparse_p
  #( int wexp = 5, wsig_ac = 10, wsig_in = 6, wsig_wd = 4 )
   ( output logic done, input uwire start );

   localparam int ni = 4;
   localparam int wo = 1 + wexp + wsig_ac;
   localparam int wi = 1 + wexp + wsig_in;
   localparam int ww = ni * ( 1 + wexp + wsig_wd );

   localparam int wsig_ws = 2 * wsig_wd + wexp + 1;
   localparam int ws = 1 + wexp + wsig_ws;
   localparam int wd = 1 + wexp + wsig_wd;

   localparam real tol_s = real'(2) / ( 1 << min(wsig_in,wsig_ws) );
   localparam real tol_d = real'(2) / ( 1 << wsig_wd );
```

```
localparam int n_tests = 5000;
localparam real hot_val[] = { 1, 2, 0.1, 10.1 };
localparam int n_one_hot = 4;
localparam int n_two_hot = n_one_hot;
initial if ( n_one_hot != hot_val.size() )
  $fatal(1,"Fix n_one_hot and file a Cadence bug.");

logic [wo-1:0] o;
logic [wi-1:0] ia[ni];
logic [ww-1:0] wht;
logic [ni-1:0] fmt;

localparam logic [5:0][3:0] fmts =
          { 4'b11, 4'b110, 4'b1100, 4'b101, 4'b1010, 4'b1001 };

nn_sparse #(ni, wexp, wsig_ac, wsig_in, wsig_wd) nnsp(o, ia, wht, fmt);

initial begin

   automatic int n_errd = 0, n_errs = 0;
   automatic real max_diffs = 0, max_diffd = 0;
   automatic string abbrev =
          $sformatf("ex%0d,ac%0d,in%0d,wd%0d",wexp,wsig_ac,wsig_in,wsig_wd);
   wait ( start );
   $write("Testing %s: wexp=%0d, wsig_ac=%0d, wsig_in=%0d, wsig_wd=%0d\n",
          abbrev, wexp, wsig_ac, wsig_in, wsig_wd);

   for (int tn = 0; tn < n_tests; tn++ ) begin

      automatic int sidx = 0;
      automatic int hot = tn % 4;
      automatic int rnd = tn / 4;
      automatic int one_hot = rnd < n_one_hot;
      automatic int two_hot = !one_hot && rnd - n_one_hot < n_two_hot;
      automatic int sparse = one_hot || two_hot || {$random} & 1;

      automatic int h2 = ( hot + 1 + {$random}%3 ) % 4;

      real shadow_ia[4], shadow_w[4], shadow_o, diff, oreal, tol;
      real max_diff;
      logic [3:0][wd-1:0] wht4;
      logic [1:0][ws-1:0] wht2;
      fmt = one_hot || two_hot ? ( 1<<hot ) | ( 1<<h2 )
      : sparse ? fmts[{$random}%6] : 4'hf;
      tol = sparse ? tol_s : tol_d;
      shadow_o = 0;
      for ( int i=0; i<4; i++ ) begin
         automatic real iav = real'({$random}) / ( 1 << 30 );
         automatic real w = real'({$random}) / ( 1 << 30 );
         if ( one_hot || two_hot )
           begin
              iav = 1.0 + real'(i)/10;
              w = i == hot || two_hot && i == h2 ? hot_val[rnd] : 0;
```

```
            end
          shadow_w[i] = w;
          shadow_ia[i] = iav;
          wht4[i] = conv#(wexp,wsig_wd)::rtof(w);
          ia[i] = conv#(wexp,wsig_in)::rtof(iav);
          if ( sparse && fmt[i] ) wht2[sidx++] = conv#(wexp,wsig_ws)::rtof(w);
          if ( fmt[i] ) shadow_o += iav * w;
       end
       wht = sparse ? wht2 : wht4;
       #1;
       oreal = conv#(wexp,wsig_ac)::ftor(o);
       diff = fabs( shadow_o - oreal ) / fabs( shadow_o ? shadow_o : 1 );
       max_diff = sparse ? max_diffs : max_diffd;

       if ( ! ( diff < tol ) ) begin
          automatic int n_err = sparse ? ++n_errs : ++n_errd;
          if ( n_err < 5 || 0 && diff > max_diff ) begin
             automatic int ilast = fmt[3] ? 3 : fmt[2] ? 2 : 1;
             $write( "Error tn=%0d for fmt %4b  %h = %7.4f != %7.4f (correct)\n",
                     tn, fmt, o, oreal, shadow_o );
             $write( "      ");
             for ( int i=0; i<4; i++ )
               if ( fmt[i] )
                 $write( "%.4f %.4f%s", shadow_ia[i], shadow_w[i],
                         i < ilast ? " + " : "\n");
             $write( "      ");
             for ( int i=0; i<4; i++ )
               if ( fmt[i] )
               $write( "%.4f     %s", shadow_ia[i] * shadow_w[i],
                        i < ilast ? " + " : "\n");
             if ( 0 )
               $write( "      diff %.8f,  tol %.8f\n",diff,tol);

             // Feel free to modify or add to this to help with your solution.
             $write( "      acc1 = %h = %.4f\n",
                      nnsp.acc1, conv#(wexp,wsig_ac)::ftor(nnsp.acc1));

          end
       end

       if ( diff > max_diff ) begin
          if ( sparse ) max_diffs = diff; else max_diffd = diff;
       end

    end

    $write("Done with %s %0d tests, %0d, %0d  sp, den errors found.\n",
           abbrev, n_tests, n_errs, n_errd);
    $write("For %s  max diff %f, %f  sp, den.\n",
           abbrev, max_diffs, max_diffd);
    done = 1;

  end
```

```
endmodule


module testbench_hy;

   localparam int n_tests = 5;

   localparam int w_sig_a = 10;
   localparam int w_sig_b = 20;
   localparam int w_sig_p = 25;
   localparam int w_exp = 5;
   localparam int wa = 1 + w_exp + w_sig_a;
   localparam int wb = 1 + w_exp + w_sig_b;
   localparam int wp = 1 + w_exp + w_sig_p;
   localparam int bias_hy = ( 1 << w_exp - 1 ) - 1;
   localparam int bias_sr = ( 1 << 8 - 1 ) - 1;
   localparam int bias_r = ( 1 << 11 - 1 ) - 1;
   localparam int w_sig_r = 52;
   localparam int w_exp_r = 11;
   localparam int w_sig_min = min3( w_sig_a, w_sig_b, w_sig_p );
   localparam real tol = 1.0 / ( longint'(1) << w_sig_min );

   logic [wa-1:0] a;
   logic [wb-1:0] b;
   uwire [wp-1:0] prod;

   hy_mult #(w_exp,w_sig_a,w_sig_b,w_sig_p) hm1(prod,a,b);

   initial begin

      automatic int n_err = 0;
      automatic real diff_max = 0;

      for (int i=0; i<n_tests; i++ ) begin

         automatic real a_shadow = real'($random()) / (1<<31);
         automatic real b_shadow = real'($random()) / (1<<31);
         automatic real prod_correct = a_shadow * b_shadow;
         real prodf, diff;

         a = conv#(w_exp,w_sig_a)::rtof(a_shadow);
         b = conv#(w_exp,w_sig_b)::rtof(b_shadow);

         #1;

         prodf = conv#(w_exp,w_sig_p)::ftor( prod );
         diff = fabs( prodf - prod_correct );
         if ( diff > diff_max ) diff_max = diff;

         if ( ! ( diff < tol ) ) begin
            n_err++;
            if ( n_err < 4 )
              $write( "Error for %.3f * %.3f:  %.4f != %.4f (correct)\n",
                      a_shadow, b_shadow, prodf, prod_correct);
```

```
         end

      end

      $write("Done with hy %d tests, %d errors found. Max diff %f\n",
             n_tests, n_err, diff_max);

   end

endmodule

// cadence translate_on

`default_nettype wire
`include "/apps/linux/cadence/GENUS211/share/synth/lib/chipware/sim/verilog/CW/CW_fp_mult.v"
`include "/apps/linux/cadence/GENUS211/share/synth/lib/chipware/sim/verilog/CW/CW_fp_add.v"
```

LSU EE 4755 Homework 3 Solution Due: 18 October 2021

To help solve the problems below, look at problems listed in the simple model slides, 2020 Homework 4, 2019 Midterm Exam Problem 2b and c, and especially 2018 Final Exam problems 1 and 2.

Problem 1: As requested in the subproblems below use the simple model to determine the cost and delay of the `insert_at` module from the solution to Homework 1 (see last page) instantiated with `wa` $= w_a$ and `wb` $= w_b$, and using $C_{\text{lsb}}(w_a)$ for the cost of the `mask_lsb` module and $D_{\text{lsb}}(w_a)$ for the delay of the `mask_lsb` module. The `wo` and `walg` parameters are not set so you can use their default values, $w_o = w_a + w_b$, $l_a = \lceil \lg(w_a + 1) \rceil$, and $l_b = \lceil \lg w_b \rceil$, in your answers.

For partial credit, and to help you solve the problems provide a sketch of the inferred hardware. It may help to first solve the problem for specific values of w_a and w_b, and then to generalize for arbitrary w_a and w_b.

(a) Find the cost and delay of the hardware inferred for the line of Verilog from `insert_at` shown below. Just for the hardware described on the line. There's no trick, this part is easy. Just remember to express your answers in terms of w_a, w_b, and w_o.

```
assign o = ia_high | ib_at_pos | ia_low;
```

Suppose for a moment that each of the quantities being ORd, `ia_high`, `ib_at_pos`, and `ia_low`, are w_o bits. Then for each of the w_o bit positions in `o` there will be a 3-input OR gate (or possibly two 2-input OR gates) and the total cost would be $2w_o\,\text{u}_\text{c}$. But while `ia_high` and `ib_at_pos` are w_o bits, `ia_low` is only w_a bits. So the cost of the hardware computing the low w_a bits of `o` will be $2w_a\,\text{u}_\text{c}$. Each of the remaining $w_o - w_a = w_b$ bits will just be an OR of a bit of `ia_high` with a bit of `ib_at_pos`, for a cost of $w_b\,\text{u}_\text{c}$. So the total cost will be $\boxed{[2w_a + w_b]\,\text{u}_\text{c}}$ or equivalently $\boxed{[w_o + w_a]\,\text{u}_\text{c}}$.

The low w_a bits are computed using either two 2-input OR gates or a 3-input OR gate, either way $\boxed{\text{the delay is } 2\,\text{u}_\text{t}}$. Note that the delay should be based on the critical path, and in this case it is one of the low w_a bits. I suppose it's nice that those other bits are computed in just $1\,\text{u}_\text{t}$ but the important number is when all bits are done.

Grading Note: Many gave the delay as $\lceil \lg 3 \rceil\,\text{u}_\text{t}$. Normally I don't expect numbers to be computed for arithmetic expressions, but that's for complex ones. In this case, please just give the answer as 2, lest I assume you don't know what $\lceil \lg 3 \rceil\,\text{u}_\text{t}$ means.

Common Mistake: A common mistake was to OR together all $2w_o + w_a$ bits in one big OR gate, or perhaps two large OR gates. That's wrong because that's not what a bitwise OR does.

(b) Find the cost and delay of the `shift_left` module instances `slc` and `slb` taking into account any constant inputs and assuming that the synthesis program infers a logarithmic shifter. Don't forget that your answer must be in terms of w_a, w_b, w_o, l_a, and l_b, and that these denote the parameters of `insert_at`, not the parameters of the shifters. For more information on the logarithmic shifter see the additional material provided for the Set 1 lectures on the course lectures page.

Before cutting-and-pasting simple-model cost and delay expressions for a logarithmic shifter, take a close look at the parameters set for `slc` and `slb` and be sure to optimize for them. Notice that unlike typical shifters, the shift-out and shift-in ports are not the same size and that the shift amount is not necessarily ceiling-log-two of the input width.

Hint: The cost and delay for one of these shifters will be really easy to compute.

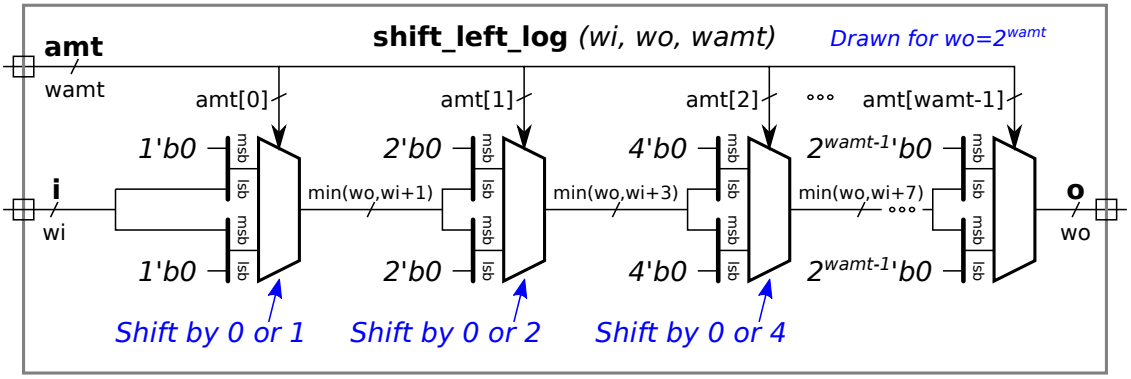
Notice that the shift amount connection (`amt`) to `slc` is an elaboration-time constant, `wb`. Therefore, the cost of `slc` is zero. A bit in the output `ia_high` is either connected to a bit of input `ia_high_low` or to the constant zero.

Grading Note: Most people did not see that the shifter required no hardware at all (other than something to generate a constant zero which would be optimized away). A few that did notice that the shift amount was zero did not properly optimize the multiplexors to which the shift amount is connected. If one of the data inputs of a mux is constant the cost drops from $3\,\mathrm{u_c}$ to $1\,\mathrm{u_c}$ per bits. But if the select input is constant the cost goes to zero. If that's not obvious please review what a mux does.

Next, consider `slb`, in which none of inputs are constant. The width of the input is w_b, the width of the output is w_o, and the input can be shifted by at most $2^{\lceil \lg(w_a+1)\rceil}$ bits. Let $l_a = \lceil \lg(w_a+1)\rceil$, that's the number of bits used to represent the shift amount. The value of the shift amount is at most $2^{l_a}-1$.

A logarithmic shifter with an l_a-bit shift amount consists of l_a multiplexors, one multiplexor for each bit in the shift amount. Multiplexor 0 shifts by either 0 or $2^0 = 1$ bit, mux 1 shifts by either 1 or $2^1 = 2$ bits, mux i shifts by either 0 or 2^i bits, and mux $l_a - 1$ shifts by 0 or 2^{l_a-1} bits. In a conventional logarithmic shifter with $l_a = 4$, the input and output would each be $2^4 = 16$ bits, and as a whole the shifter could shift by an amount ranging from 0 bits to 15 bits (but not by 16 bits). (Why not 16 bits? That's a convention, but why not allow a shift amount that would shift away all of the bits. Good question, I'm sure it was debated.)

Lets consider the shifter needed for `slb`. Let the first multiplexor making up this shifter shift by 0 or 1 bits. In a conventional shifter the mux has two w-bit inputs and a w-bit output. But in `slb` the output will be larger than the input, w_o bits. So we need to make the mux large enough to handle the largest value produced at that stage. For the first stage, since it can shift by one bit, we need to make the mux $w_b + 1$ bits (remembering that input is w_b bits). The second mux can shift by 0 or 2 bits, and to it needs to be $w_b + 1 + 2 = w_b + 3$ bits. Because the output is w_o bits the maximum mux size is w_o bits, which will be the last mux. That last mux can shift by 0 or 2^{l_a-1} bits. (Because w_a and w_b are not constrained, it is not always true that $2^{l_a-1} = w_a/2$.) The diagram below shows such a shifter in which `wi` would be used for w_b and `wamt` would be used for l_a.



A general w-bit 2-input mux has cost $3w\,\mathrm{u_c}$. But in a shifter some mux input bits are zero, and at those positions the cost is $1\,\mathrm{u_t}$ each. First lets assume that all bits have cost $w\,\mathrm{u_t}$. Also, lets restrict ourselves to the case where $w_o = w_b + 2^{l_a-1}$.

The cost under that assumption and restriction is

$$
\begin{aligned}
C_{\mathrm{sl-noopt}}(w_b, w_o, l_a) &= \sum_{i=0}^{l_a-1} 3\left(w_b + \sum_{j=0}^{i} 2^j\right)\,\mathrm{u_c} \\
&= \sum_{i=0}^{l_a-1} 3(w_b + 2^{i+1} - 1)\,\mathrm{u_c} \\
&= \left[3(w_b-1)l_a + \frac{3}{2}(2^{l_a}-1)\right]\,\mathrm{u_c}
\end{aligned}
$$

For a tighter cost estimate, consider the number of zero bits in stage i. Stage i shifts by 2^i bits and so 2^i zeros must be appended to the most-significant side of the unshifted input and 2^i zeros are appended to the least-significant side of the shifted input. So there are 2×2^i mux bits with a zero at either input, and so the cost is

$[3(w_b + 2^{i+1} - 1) - 2 \times 2 \times 2^i]\,\mathrm{u_c}$ or $[3(w_b + 2^{i+1} - 1) - 2 \times 2^{i+1}]\,\mathrm{u_c}$ or $[3(w_b - 1) + (3-2)2^{i+1}]\,\mathrm{u_c}$ or $[3(w_b - 1) + 2^{i+1}]\,\mathrm{u_c}$.

The total cost is

$$\begin{aligned} C_{\mathrm{sl-opt}}(w_b, w_o, l_a) &= \sum_{i=0}^{l_a-1} [3(w_b - 1) + 2^{i+1}]\,\mathrm{u_c} \\ &= [3(w_b - 1)l_a + \frac{1}{2}(2^{l_a} - 1)]\,\mathrm{u_c} \end{aligned}$$

Grading Note: No one computed the cost completely correctly. A small deduction, 0.5, was given for a cost of $w_o l_a\ \mathrm{u_c}$ since that overstates the cost of all but the last mux. A much larger deduction was given if the cost was based on muxen that were too small.

The delay is far less tedious to compute because regardless of the size of each multiplexor, the critical path through a mux passes through two 2-input gates. Under the simple model their delay is $2\,\mathrm{u_t}$, and so the total delay is $2l_a\ \mathrm{u_t}$. That's it.

(*c*) Find the cost and delay of `insert_at`. Use the answers above and work out cost and delay for the remaining hardware in the module. Don't forget to use $C_{\mathrm{lsb}}(w_a)$ for the cost of the `mask_lsb` module and $D_{\mathrm{lsb}}(w_a)$ for the delay of the `mask_lsb` module.

For this discussion refer to the `insert_at` module below which includes labels such as `Line 1` in the comments. In the sub-problems above the cost and delay of hardware described by Lines 7, 5, and 6 has been found. The cost and delay of the `ml` instance, Line 1, are given in this problem as $C_{\mathrm{lsb}}(w_a)$ and $D_{\mathrm{lsb}}(w_a)$. The Verilog on Line 4 is executed at elaboration time and so does not describe hardware. All that remains to work out is the hardware described on Lines 2 and 3.

Each of these lines is a bitwise AND of two w_a-bit quantities, for a cost of $w_a\,\mathrm{u_c}$ each. Their delay is $1\,\mathrm{u_t}$. Combining all of these yields the total cost,

$$C_{\mathrm{insertat}}(w_a, w_b) = \big[\ \overbrace{C_{\mathrm{lsb}}(w_a)}^{\texttt{ml}-\mathrm{L1}} + \overbrace{2w_a}^{\mathrm{L2\text{-}3}} + \overbrace{0}^{\mathrm{L5}} + \overbrace{3(w_b-1)l_a + \frac{1}{2}(2^{l_a}-1)}^{\texttt{slb}-\mathrm{L6}} + \overbrace{2w_a + w_b}^{\texttt{o}-\mathrm{L7}}\ \big]\,\mathrm{u_c}$$

Collecting terms and using C_{lsb} from the problem below:

$$\begin{aligned} C_{\mathrm{insertat}}(w_a, w_b) &= \left[C_{\mathrm{lsb}}(w_a) + 2w_a + 0 + 3(w_b-1)l_a + \frac{1}{2}(2^{l_a}-1) + 2w_a + w_b\right]\,\mathrm{u_c} \\ &= [w_a + 2^{l_a} - 4 + 2w_a + 0 + 3(w_b-1)l_a + \frac{1}{2}(2^{l_a}-1) + 2w_a + w_b]\,\mathrm{u_c} \\ &= [w_a + w_a - 4 + 2w_a + 0 + 3(w_b-1)l_a + \frac{1}{2}(w_a-1) + 2w_a + w_b]\,\mathrm{u_c} \\ &= [6.5w_a - 4.5 + 3(w_b-1)l_a + w_b]\,\mathrm{u_c} \\ &= [3(w_b-1)l_a + w_b + 6.5w_a - 4.5]\,\mathrm{u_c} \end{aligned}$$

The dominant term is $3w_b l_a$, which isn't so bad.

```
// SOLUTION -- Line numbers are referenced in the solution discussion.
module insert_at #( int wa = 20, wb = 10, wo = wa+wb, walg = $clog2(wa+1) )
   ( output logic [wo-1:0] o,
     input uwire [wa-1:0] ia, input uwire [wb-1:0] ib,
     input uwire [walg-1:0] pos );

   uwire [wa-1:0] mask_low;
   mask_lsb #(wa) ml(mask_low, pos);              // Line 1.
   uwire [wa-1:0] ia_low = ia & mask_low;         // Line 2.
   uwire [wa-1:0] ia_high_low = ia & ~mask_low;   // Line 3.

   localparam int wblg = $clog2(wb);  // Line 4. No Hardware. (Computed during elaboration.)
   uwire [wo-1:0] ia_high;
   shift_left #(wa,wo,wblg) slc( ia_high, ia_high_low, wblg'(wb) ); // Line 5

   uwire [wo-1:0] ib_at_pos;
   shift_left #(wb,wo,walg) slb( ib_at_pos, ib, pos );              // Line 6

   assign o = ia_high | ib_at_pos | ia_low;                         // Line 7

endmodule
```

To find the total delay **we need to find the critical path**. *Note: Emphasis added after grading.* The critical path is easy to find because the parts taking a substantial amount of time, `ml` (the `mask_lsb` instance) and `slb`, connect only to `insert_at` module inputs. The default assumption for timing analysis is that module inputs arrive at $t = 0$, and so the output of `ml` is available at $D_{\text{lsb}}(w)$ and the output of `slb` is available at $2l_a$ u_t. Peeking ahead to the solution of the next problem, we know that `ml` has a delay of l_a u_t.

The output of both `ml` and `slb` each connect only to the `o` expression, Line 7, and so the critical path is from `slb` to Line 7. That would add a delay of 1 (if connected intelligently), and so the delay is $D_{\text{insertat}}(w_a, w_b) = [2l_a+1]$ u_t, where $l_a = \lceil \lg(w_a + 1) \rceil$.

Problem 2: Some of you may have seen this coming: Find expressions for $C_{\text{lsb}}(w)$, the cost of the `mask_lsb` module and $D_{\text{lsb}}(w)$, the delay of the `mask_lsb` module, in both cases `wo` $= w$, where `wo` is the parameter used in the `mask_lsb` definition. Assume a well-optimized design, not something that uses w $\lceil \lg w \rceil$-bit magnitude comparison units.

Hint: Think about the problem for about 30 minutes, then look at 2018 Final Exam Problems 1 and 2.

The `gtd_rec` module from the 2018 final exam is similar to `mask_lsb` but has three differences. In `mask_lsb` the input value, `n1`, specifies that there should be `n1` ones followed by zeros. In `gtd` the input value, `iter`, specifies that there should be `iter+1` zeros followed by ones. The second difference (or a consequence of the first) is that while the output of `mask_lsb` can be all zeros or all ones, the output of `gtd_rec` must have at least one zero. Finally, `gtd_rec` can only be instantiated at power-of-two sizes.

Those minor differences are easy to fix. For example, inverting the output (change each zero to a one) will fix the first difference. The non-power-of-two issue can be fixed by making sure that the size of the recursive instantiation is always a power of two. The initial instantiation does not have to be a power of two. Also a special case can be added to the initial instantiation to handle the all ones case.

I'm tempted to show the recursive version of `mask_lsb`, but I might make it a midterm exam problem. (Not the whole thing, just a small part.) If I do I'll provide a warning in class on Monday, 25 October 2021.

For cost, the easiest thing to do is assume that w is a power of 2 and then just use the expressions from the exam. Using this assumption: $C_{\mathrm{lsb}}(w) = [2w - 4]\,\mathrm{u_c}$. For arbitrary positive w the cost of the initial instantiation is $w\,\mathrm{u_c}$ and the cost of the recursive instantiation (one level down) is $2^{\lceil \lg w \rceil - 1}\,\mathrm{u_c}$. The terminal case for recursion is for $w = 2$, and the cost of that hardware is zero under the simple model. So the summation will end at $w = 4$ (which is $i = 2$ in the summation). The total cost is

$$\begin{aligned} C_{\mathrm{lsb}}(w) &= [w + \sum_{i=l_w-1}^{2} 2^i]\,\mathrm{u_c} \\ &= [w + 2^{l_w} - 4]\,\mathrm{u_c} \end{aligned}$$

where $l_w = \lceil \lg w \rceil$.

Each level has a delay of 1, and so the total delay is $[\lceil \lg w \rceil - 1]\,\mathrm{u_t}$ for $w \geq 4$.

An uncommented Homework 1 solution appears below.
For the full version visit `https://www.ece.lsu.edu/koppel/v/2021/hw01-sol.v.html`.

```
module insert_at
  #( int wa = 20, wb = 10, wo = wa+wb, walg = $clog2(wa+1) )
   ( output logic [wo-1:0] o,
     input uwire [wa-1:0] ia,
     input uwire [wb-1:0] ib,
     input uwire [walg-1:0] pos );

   uwire [wa-1:0] mask_low;
   mask_lsb #(wa) ml(mask_low, pos);
   uwire [wa-1:0] ia_low = ia & mask_low;
   uwire [wa-1:0] ia_high_low = ia & ~mask_low;

   localparam int wblg = $clog2(wb);
   uwire [wo-1:0] ia_high;
   shift_left #(wa,wo,wblg) slc( ia_high, ia_high_low, wblg'(wb) );

   uwire [wo-1:0] ib_at_pos;
   shift_left #(wb,wo,walg) slb( ib_at_pos, ib, pos );

   assign o = ia_high | ib_at_pos | ia_low;

endmodule

module shift_left
  #( int wi = 4, wo = wi, wolg = $clog2(wo) )
   ( output uwire [wo-1:0] o,
     input uwire [wi-1:0] i,
     input uwire [wolg-1:0] amt );
   assign o = i << amt;
endmodule

module mask_lsb
  #( int wo = 6, wp = $clog2(wo+1) )
   ( output logic [wo-1:0] o, input uwire [wp-1:0] n1 );
   always_comb for ( int i=0; i<wo; i++ ) o[i] = i < n1;
endmodule
```

LSU EE 4755 Homework 4 Solution Due: 11 November 2021

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/2021/hw04.v.html`.

Problem 0: If necessary, follow the instructions at `https://www.ece.lsu.edu/koppel/v/proc.html` to set up your class account, copy the assignment, and run the Verilog simulator and synthesis program on the unmodified homework file, `hw04.v`. Do this early enough so that minor problems (*e.g.*, password doesn't work) are minor problems.

Teamwork

Students can work on this assignment in teams. Each student should submit his or her own assignment but list team members. It is recommended that one team member be responsible for learning SimVision.

Every member of a team that has completed a project, must be capable of re-solving the problem. It is recommended that all team members re-solve the problem on their own for their own pedagogical benefit.

Problem 1: Module `bit_keeper` has a w_b-bit output `bits` ($_b$ is for width of buffer) and a 1-bit output `ready`. Think of output `bits` as a long bit vector (w_b bits long) that is edited using the module's inputs. Commands to edit `bits` are given using four-bit input `cmd` (command), w_i-bit input `din` (data in), and w_s-bit input `pos` (position). The module is to operate sequentially using input `clk`.

Complete `bit_keeper` as described below, and make sure that it is synthesizable. As always, code should be written clearly, and designs should not be costly or slow.

When completed `bit_keeper` should operate as follows. On a positive edge of `clk` action is taken based on the value of `cmd`. The possible values of `cmd` are: `Cmd_Reset`, `Cmd_None`, `Cmd_Write`, and `Cmd_Rot_To`. (These can be used as constants in your code. The constants are defined by `enum Command`.) Some commands will be complete in one cycle (the cycle in which the `cmd` is set up to the positive edge of `clk`). Other commands will take multiple cycles.

Be sure to understand the details of how multi-cycle commands execute. When a multi-cycle command starts the `ready` output must be set to zero and must be held at zero until the command completes. The command and its arguments will only be held at the inputs **for one cycle**, and so at the next positive clock edge they will be gone. The `cmd` input will be set to `Cmd_Nop`, and the `pos` and `din` inputs will be set to random values. This means that the inputs of multi-cycle commands that will be needed in subsequent cycles must be saved in registers.

The testbench can emit a trace of commands and their effects. This trace is used below to illustrate what the module is supposed to do. The trace is collected after the command completes. A trace entry starts with the word `Cycle`. The cycle number is shown, followed by command details, followed by the state of bits.

For `Cmd_Reset` output `bits` should be set to zero. Also, any internal registers should be set to zero. The command should complete at the positive edge. This should set `ready` to 1. In the trace below the reset command set bits back to zero. Notice that the command completes in one cycle (based on the cycle numbers).

```
Cycle 307 -- test 73: Cmd_Nop                    : bits = 01401f4
Cycle 308 -- test 74: Cmd_Reset                  : bits = 0000000
```

For `Cmd_Rot_To` the value in `bits` must be rotated so that the contents of `bits[0]` is moved to `bits[pos]`, `bits[1]` is moved to `bits[(pos+1)%wb]`, and so on. This is like a left shift of `pos` bits, except that the most significant `pos` bits of `bits` are rotated into the the `pos` least significant bits. In the trace below the rotate command rotates four bits (one hexadecimal digit). Notice that the most-significant digit on the first line is rotated to the least significant digit after the rotation command.

```
Cycle 301 -- test 71: Cmd_Nop                    : bits = 401401f
Cycle 306 -- test 72: Cmd_Rot_To pos 4           : bits = 01401f4
```

This rotation **must be performed** using two instances of module `rot_left`. One instance should rotate by 1, the other rotates by a larger value, call it r_b, of your choosing. Each clock cycle the value of `bits` is rotated using one of these, but never both in the same clock cycle. Use the r_b-bit rotate instance until the number of bit positions to shift is $\leq r_b$, then use the 1-bit rotate instance.

Command `Cmd_Write` has two forms based on the value of input `pos`. If `pos` is zero then the least significant w_b bits of `bits` should be written with `din`. This should complete at the positive edge. Otherwise, bits `pos` through `pos+wi-1` of `bits` should be written with `din`—but not directly. Instead, `bits` should be rotated so that bit `pos` is at the least-significant position, then the data should be written, then `bits` should be rotated back to its original position. Use only the two `rot_left` instances.

The trace below shows a write with `pos=0`:

```
Cycle 417 -- test 86: Cmd_Nop                    : bits = 0000240000
Cycle 418 -- test 87: Cmd_Write  pos 0, data 7   : bits = 0000240007
```

When `pos` is non-zero the writes take longer:

```
Cycle  96 -- test 20: Cmd_Nop                    : bits = 0a0000003c
Cycle 107 -- test 21: Cmd_Write  pos 27, data 4  : bits = 0a2000003c
```

No action is needed for command `Cmd_Nop`. In fact, this is the command that will be present while the external hardware, including the testbench, is waiting for other commands to complete.

The testbench will test `bit_keeper` at two sizes. At each size detailed information is given for the first few errors. That includes a trace of commands leading up to the error, followed by the erroneous command, and what the `bits` should have been. After each error the testbench sets its shadow value of `bits` to the erroneous output so that subsequent tests can pass. Here is in example of the output:

```
Cycle  22 -- test  0: Cmd_Rot_To pos 20          : bits = 0000000000
Cycle  54 -- test  1: Cmd_Rot_To pos 31          : bits = 0000000000
Cycle  55 -- test  2: Cmd_Nop                    : bits = 0000000000
Cycle  96 -- test  3: Cmd_Write  pos 37, data 2  : bits = 4000000000
Cycle  97 -- test  4: Cmd_Nop                    : bits = 4000000000
Cycle 103 -- test  5: Cmd_Rot_To pos 5           : bits = 0000000008
Cycle 104 -- test  6: Cmd_Write  pos 0, data 3   : bits = 0000000003
Error in test  7: Cmd_Write  pos 1, data 2   : 0000000c04 != 0000000005 (correct)
```

For multi-cycle commands the testbench will wait for `ready` to go to zero and then back to one. If that does not happen after a certain number of cycles the testbench will *timeout*, meaning that it will give up waiting and print a `CYCLE LIMIT EXCEEDED` message. If there is a timeout while a command is in progress (meaning that `ready` did go to zero, but did not return to one) the testbench will show a trace of recent history, followed by an indication of what it was waiting for:

```
Exit from clock loop at cycle 16000, limit 16000, ** CYCLE LIMIT EXCEEDED **
```

```
** Preceding Commands **
Cycle   7 -- test  0: Cmd_Rot_To pos 20          : bits = 0000000000
Cycle  14 -- test  1: Cmd_Rot_To pos 31          : bits = 0000000000
Cycle  15 -- test  2: Cmd_Nop                    : bits = 0000000000

** In-Progress Command **
test  3: Cmd_Write  pos 37, data 2
 -- Awaiting ready = 1.
```

If the testbench does not timeout then it will print a tally of the number of errors after testing each `bit_keeper` instance. Also, as a measure of quality, the testbench reports the average number of cycles to perform `Cmd_Rot_To` and `Cmd_Write` (with non-zero `pos`). For example,

```
Starting tests for (wb=40,wi=4)
Finished 200 tests for (wb=40,wi=4), 0 errors.
Avg cyc  Cmd_Rot_To 5.5 (67)  Cmd_Write 10.6 (35)

Starting tests for (wb=28,wi=8)
Finished 140 tests for (wb=28,wi=8), 0 errors.
Avg cyc  Cmd_Rot_To 4.2 (57)  Cmd_Write 8.2 (18)
```

The lines starting `Avg cyc` report timing. The number in parentheses is the number of times the command was issued. So for the first set of tests `Cmd_Rot_To` was tried 67 times, and the average number of cycles taken to complete it was 5.5.

A lower number for `Avg cyc` can indicate a good design, or that certain rules were not followed.

It is very important that debugging tools are used. Take advantage of the testbench messages to see what is going wrong. Run SimVision to get a detailed look at what your module is doing.

The solution has been copied to the homework directory, and an htmlized version has been posted at `https://www.ece.lsu.edu/koppel/v/2021/hw04-sol.v.html`. For the discussion below the solution is shown in pieces, shorn of most comments. Following that is the complete solution. The solution starts by specifying rotate amounts for the two rotation modules, followed by their instantiation.

```
localparam int rot_amt_a = 1;
localparam int rot_amt_b = 1 << ( ws >> 1 );

uwire [wb-1:0] ra, rb;
rot_left #(wb,rot_amt_a) rl1(ra,bits);
rot_left #(wb,rot_amt_b) rl8(rb,bits);
```

The rotate amount of the first module is set to 1, but a localparam is used for its value. To minimize the number of rotations the rotate amount for the second module, `rot_amt_b`, should be set to the square root of `wb`. To minimize delay it should be set to a power of 2. Here it is set to a power of 2 close to the square root of `wb`.

Rotations are to be done over several cycles. As stated in the problem commands are presented at the inputs for just for one cycle, and are then replaced with a `Cmd_Nop` until the `ready` returns to 1. To remember what needs to be done three registers will be used, `rot_to_do`, `rot_to_return`, and `wval`. Register `rot_to_do` is set to the number of bits of rotation that still need to be done. For `Cmd_Rot_To` it is initialized to `pos` and for `Cmd_Write` with `pos!=0` it is initialized to `wb - pos`. Register `rot_to_return` is set to the amount of rotation needed after the write is performed. Register `wval` is the value to write.

The `ready` output is set to 1 when both `rot_to_do` and `rot_to_return` are both zero.

```
logic [ws-1:0] rot_to_do;      // Remaining amount of rotation to do.
logic [ws-1:0] rot_to_return;  // Amount of rotation needed after write.
```

```
logic [wi-1:0] wval;           // Value to write.
assign ready = rot_to_do == 0 && rot_to_return == 0;
```

The main always_ff has just a single case statement. Cmd_Reset is straightforward:

```
always_ff @( posedge clk ) begin
   case ( cmd )

     Cmd_Reset: begin
        bits = 0;
        rot_to_do = 0;
        rot_to_return = 0;
     end
```

For Cmd_Rot_To the rotate amount is saved in rot_to_do. The work of rotating is done when cmd is Cmd_Nop.

```
     Cmd_Rot_To: begin rot_to_do = pos; end
```

What Cmd_Write does depends on pos. If it's zero the write is done immediately. Otherwise rot_to_do is set to an amount that will bring bit pos to the least-significant position. Variable rot_to_return is set to the rotation to use after the write completes, one which moves the least-significant bit back to where it was. Also, the write value is saved.

```
     Cmd_Write:
       if ( pos == 0 ) begin
          bits[wi-1:0] = din;
       end else begin
          rot_to_do = wb - pos;
          wval = din;
          rot_to_return = pos;
       end
```

The work of rotating is done when cmd is set to Cmd_Nop. If rot_to_do is non-zero (which means $\geq$ rot_amt_a) then bits is set to the output of the appropriate rotation module and rot_to_do is decremented. Note that the rotation being performed can be for one of three purposes: a Cmd_Rot_To, the rotation before a write, or the rotation after a write.

```
     Cmd_Nop: begin
        if ( rot_to_do >= rot_amt_b ) begin
           bits = rb;                // Use output of larger rot module.
           rot_to_do -= rot_amt_b;   // Decrement remaining rot amt.
        end else if ( rot_to_do >= rot_amt_a ) begin
           bits = ra;                // Use output of smaller rot module.
           rot_to_do -= rot_amt_a;   // Decrement remaining rot amt.
        end
      // More Cmd_Nop code below
```

Next, Cmd_Nop needs to check whether a write needs to be done now. (A write needs to be done if rot_to_return is non-zero and it needs to be done now if also rot_to_do is zero.) If so, the write is performed and rot_to_do is set so that bits is rotated back to its original position.

```
        if ( rot_to_do == 0 && rot_to_return !=0 ) begin
           bits[wi-1:0] = wval;
           rot_to_do = rot_to_return;
           rot_to_return = 0;
```

```
        end
```

The entire solution with more comments appears below.

Grading Notes: In many solutions there were three separate pieces of code to perform rotate: one used for `Cmd_Rot_To`, one used before a write, and one used after a write. That code duplication makes it harder for humans to read, and could also lead to more costly and slower designs.

```
module bit_keeper
  #( int wb = 64, wi = 8, ws = $clog2(wb) )
   ( output logic [wb-1:0] bits,
     output uwire ready,
     input uwire [3:0] cmd,
     input uwire [wi-1:0] din,
     input uwire [ws-1:0] pos,
     input uwire clk );

   /// SOLUTION

   // Specify Rotation Amounts
   //
   localparam int rot_amt_a = 1;
   localparam int rot_amt_b = 1 << ( ws >> 1 );
   //
   // To minimize the number of rotations, rot_amt_b should be set to
   // the square root of wb. But, to minimize delay it should be set
   // to a power of 2. Here it is set to a power of 2 close to the
   // square root of wb.

   // Instantiate Rotation Modules
   //
   uwire [wb-1:0] ra, rb;
   rot_left #(wb,rot_amt_a) rl1(ra,bits);
   rot_left #(wb,rot_amt_b) rl8(rb,bits);

   logic [ws-1:0] rot_to_do;      // Remaining amount of rotation to do.
   logic [ws-1:0] rot_to_return;  // Amount of rotation needed after write.
   logic [wi-1:0] wval;           // Value to write.

   // The module is ready if there is no remaining rotation to do.
   //
   assign ready = rot_to_do == 0 && rot_to_return == 0;
```

```
always_ff @( posedge clk ) begin

   case ( cmd )

     Cmd_Reset: begin
        //
        // Perform Reset

        bits = 0;
        rot_to_do = 0;
        rot_to_return = 0;
     end

     Cmd_Rot_To: begin
        //
        // Set Amount of Rotation
        //
        // The rotation will be performed in subsequent cycles.

        rot_to_do = pos;
     end

     Cmd_Write:

       if ( pos == 0 ) begin
          //
          // Perform Write Immediately

          bits[wi-1:0] = din;

       end else begin
          //
          // Perform Write Later

          // Set amount of rotation needed before the write, ..
          //
          rot_to_do = wb - pos;
          //
          // .. save the value that will be written, ..
          //
          wval = din;
          //
          // .. and save the amount of rotation needed after the write.
          //
          rot_to_return = pos;

       end
```

```
            Cmd_Nop: begin
               //
               // Continue Executing a Cmd_Rot_To or Cmd_Write.

               // If necessary, set bits to a rotated value.
               //
               if ( rot_to_do >= rot_amt_b ) begin
                  //
                  // Still need to rotate by at least rot_amt_b bits.

                  bits = rb;                // Use output of larger rot module.
                  rot_to_do -= rot_amt_b;   // Decrement remaining rot amt.

               end else if ( rot_to_do >= rot_amt_a ) begin
                  //
                  // Still need to rotate by at least rot_amt_a (1) bit.

                  bits = ra;                // Use output of smaller rot module.
                  rot_to_do -= rot_amt_a;   // Decrement remaining rot amt.
               end

               // Check whether a write is pending and can now be performed.
               //
               if ( rot_to_do == 0 && rot_to_return !=0 ) begin
                  //
                  // Write value, and set amount of rotation to return to
                  // original positioning.

                  bits[wi-1:0] = wval;
                  rot_to_do = rot_to_return;
                  rot_to_return = 0;
               end

            end

         endcase

      end

   endmodule
```

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2021 Homework 4
//
/// SOLUTION

 /// Assignment https://www.ece.lsu.edu/koppel/v/2021/hw04.pdf
 /// Solution Discussion https://www.ece.lsu.edu/koppel/v/2021/hw04_sol.pdf


`default_nettype none


////////////////////////////////////////////////////////////////////////////////
/// Problem 1
//
 ///  Complete bit_keeper so that it applies input commands as described
 ///  in the handout.
 ///
//
//      [✔] Only modify module bit_keeper.
//      [✔] Instantiate two rot_left instances two rotate bits.
//      [✔] APPLY AT MOST ONE rotate per cycle.
//      [✔] ONLY WRITE DATA to the least-significant w bits.
//
//      [✔] Use SimVision to debug. Use command: xrun -gui hw04.v
//
//      [✔] Make sure that the testbench does not report errors.
//      [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//
//      [✔] Don't assume any particular parameter values.
//
//      [✔] Code must be written clearly.
//      [✔] Pay attention to cost and performance.
//
//      [ ] Students can work in teams. List team members in this file



typedef enum
  { Cmd_Reset = 0, Cmd_Nop, Cmd_Write, Cmd_Rot_To, Cmd_SIZE } Command;

module rot_left
  #( int w = 10, amt = 1 )
   ( output uwire [w-1:0] r, input uwire [w-1:0] a);
   assign r = { a[w-amt-1:0], a[w-1:w-amt] };
endmodule

module bit_keeper
  #( int wb = 64, wi = 8, ws = $clog2(wb) )
   ( output logic [wb-1:0] bits,
     output uwire ready,
     input uwire [3:0] cmd,
     input uwire [wi-1:0] din,
     input uwire [ws-1:0] pos,
```

```
    input uwire clk );

/// SOLUTION

// Specify Rotation Amounts
//
localparam int rot_amt_a = 1;
localparam int rot_amt_b = 1 << ( ws >> 1 );
//
// To minimize the number of rotations, rot_amt_b should be set to
// the square root of wb. But, to minimize delay it should be set
// to a power of 2. Here it is set to a power of 2 close to the
// square root of wb.

// Instantiate Rotation Modules
//
uwire [wb-1:0] ra, rb;
rot_left #(wb,rot_amt_a) rl1(ra,bits);
rot_left #(wb,rot_amt_b) rl8(rb,bits);

logic [ws-1:0] rot_to_do;      // Remaining amount of rotation to do.
logic [ws-1:0] rot_to_return;  // Amount of rotation needed after write.
logic [wi-1:0] wval;           // Value to write.

// The module is ready if there is no remaining rotation to do.
//
assign ready = rot_to_do == 0 && rot_to_return == 0;

always_ff @( posedge clk ) begin

   case ( cmd )

     Cmd_Reset: begin
        //
        // Perform Reset

        bits = 0;
        rot_to_do = 0;
        rot_to_return = 0;
     end

     Cmd_Rot_To: begin
        //
        // Set Amount of Rotation
        //
        // The rotation will be performed in subsequent cycles.

        rot_to_do = pos;
     end

     Cmd_Write:

       if ( pos == 0 ) begin
          //
          // Perform Write Immediately

          bits[wi-1:0] = din;
```

```
         end else begin
            //
            // Perform Write Later

            // Set amount of rotation needed before the write, ..
            //
            rot_to_do = wb - pos;
            //
            // .. save the value that will be written, ..
            //
            wval = din;
            //
            // .. and save the amount of rotation needed after the write.
            //
            rot_to_return = pos;

         end

      Cmd_Nop: begin
         //
         // Continue Executing a Cmd_Rot_To or Cmd_Write.

         // If necessary, set bits to a rotated value.
         //
         if ( rot_to_do >= rot_amt_b ) begin
            //
            // Still need to rotate by at least rot_amt_b bits.

            bits = rb;                // Use output of larger rot module.
            rot_to_do -= rot_amt_b;   // Decrement remaining rot amt.

         end else if ( rot_to_do >= rot_amt_a ) begin
            //
            // Still need to rotate by at least rot_amt_a (1) bit.

            bits = ra;                // Use output of smaller rot module.
            rot_to_do -= rot_amt_a;   // Decrement remaining rot amt.
         end

         // Check whether a write is pending and can now be performed.
         //
         if ( rot_to_do == 0 && rot_to_return !=0 ) begin
            //
            // Write value, and set amount of rotation to return to
            // original positioning.

            bits[wi-1:0] = wval;
            rot_to_do = rot_to_return;
            rot_to_return = 0;
         end

      end

   endcase

end
```

```
endmodule



////////////////////////////////////////////////////////////////////////////////
/// Testbench Code



// cadence translate_off


program reactivate
   (output uwire clk_reactive, output int cycle_reactive,
    input uwire clk, input var int cycle);
   assign clk_reactive = clk;
   assign cycle_reactive = cycle;
endprogram

module testbench;

   localparam int npsets = 2;
   localparam int pset[npsets][2] =
              '{ { 40, 4 }, { 28, 8 } };

   int t_errs;     // Total number of errors.
   initial t_errs = 0;
   final $write("Total number of errors: %0d\n",t_errs);

   uwire d[npsets:-1];    // Start / Done signals.
   assign d[-1] = 1;  // Initialize first at true.

   // Instantiate a testbench at each size.
   //
   for ( genvar i=0; i<npsets; i++ )
     testbench_n #(pset[i][0],pset[i][1]) t2( .done(d[i]), .tstart(d[i-1]) );

endmodule


module testbench_n
  #( int bsize = 40, isize = 5 )
   ( output logic done, input uwire tstart );

   localparam int bslg = $clog2(bsize);
   localparam int n_tests = bsize * 5;
   localparam int cyc_max = n_tests * bsize * 2;
   bit clk;
   int cycle, cycle_limit;
   logic clk_reactive;
   int cycle_reactive;
   reactivate ra(clk_reactive,cycle_reactive,clk,cycle);

   string cmd_str[int];
   initial begin
      cmd_str[Cmd_Reset] = "Cmd_Reset";
```

```
      cmd_str[Cmd_Nop] = "Cmd_Nop";
      cmd_str[Cmd_Write] = "Cmd_Write";
      cmd_str[Cmd_Rot_To] = "Cmd_Rot_To";
   end

   string event_trace, history_trace;

   initial begin
      clk = 0;
      cycle = 0;

      done = 0;
      cycle_limit = cyc_max;
      wait( tstart );

      fork
         while ( !done ) #1 cycle += clk++;
         wait( cycle >= cycle_limit ) begin
            $write("Exit from clock loop at cycle %0d, limit %0d, %s\n",
                   cycle, cycle_limit, "** CYCLE LIMIT EXCEEDED **");
            $write("** Preceding Commands **\n%s", history_trace);
            $write("** In-Progress Command **\n%s\n", event_trace);
         end
      join_any;

      done = 1;
   end


   uwire [bsize-1:0] bits;
   uwire rdy;
   bit [bsize-1:0] bits_shadow, bcpy;
   logic [bslg-1:0] pos;

   logic [3:0] cmd;
   logic [isize-1:0] din;

   bit_keeper #(bsize,isize) bk1(bits, rdy, cmd, din, pos, clk);

   typedef struct {int pos; int lat_cnt[int];} Lat_Range;

   Lat_Range lat_range[Cmd_SIZE][int];
   Lat_Range pos_range[Cmd_SIZE][int];

   initial begin

      automatic int n_err = 0;
      int n_cmd[Cmd_SIZE], n_cyc[Cmd_SIZE];
      int n_cycles;
      string cmd_info;
      for ( int i=0; i<Cmd_SIZE; i++ ) begin n_cmd[i] = 0; n_cmd[i] = 0; end

      cmd = Cmd_Reset;
      bits_shadow = bsize'(0);

      wait( tstart );
```

```
      $write("\nStarting tests for (wb=%0d,wi=%0d)\n",bsize,isize);

      @( negedge clk_reactive );
      @( negedge clk_reactive );
      cmd = Cmd_Nop;
      while ( rdy !== 1'b1 ) @( negedge clk_reactive );

      for ( int tn = 0; tn < n_tests; tn++ ) begin

         bit expect_rdy_0;
         logic [bslg-1:0] pos_given;
         logic [3:0] cmd_given;

         event_trace = $sformatf("test %2d: ",tn);

         cmd = {$random} % ( Cmd_SIZE - 1 ) + 1;
         if ( ( {$random} & 15 ) == 0 ) cmd = Cmd_Reset;
         pos = {$random} % (bsize-1) + 1;
         if ( cmd == Cmd_Write && ( {$random} & 1 ) == 0 ) pos = 0;
         din = {$random};
         cmd_given = cmd;
         pos_given = pos;

         event_trace = { event_trace, $sformatf("%-10s ",cmd_str[cmd]) };

         case ( cmd )

           Cmd_Reset: begin
              bits_shadow = 0; expect_rdy_0 = 0;
           end
           Cmd_Write: begin
              event_trace =
                { event_trace,
                  $sformatf("pos %0d, data %h", pos_given, din) };
              expect_rdy_0 = pos != 0;
              for ( int i=0; i<isize; i++ )
                bits_shadow[(i+pos)%bsize] = din[i];
           end
           Cmd_Rot_To: begin
              event_trace =
                { event_trace,
                  $sformatf("pos %0d", pos_given) };
              expect_rdy_0 = pos != 0;
              bcpy = bits_shadow;
              for ( int i=0; i<bsize; i++ )
                bits_shadow[(i+pos)%bsize] = bcpy[i];
           end
           Cmd_Nop: begin
              expect_rdy_0 = 0;
           end
           default begin
              $write("This can't happen.\n");
              $fatal(1);
           end
         endcase

         cmd_info = event_trace;
```

```
      event_trace = { event_trace, "\n" };

      @( negedge clk_reactive );

      // Wait for rdy to go to zero.
      if ( expect_rdy_0 )
        begin
           automatic int cyc_start = cycle;
           event_trace = { cmd_info, "\n -- Awaiting ready = 0.\n" };
           while ( rdy !== 1'b0 ) @( negedge clk_reactive );
           event_trace = { cmd_info, "\n -- Awaiting ready = 1.\n" };
           cmd = Cmd_Nop;
           pos = {$random};
           din = {$random};
           while ( rdy !== 1'b1 ) @( negedge clk_reactive );
           event_trace = { cmd_info, "\n -- About to check outputs.\n" };
           n_cycles = cycle - cyc_start;
        end else begin
           n_cycles = 0;
        end

      if ( bits_shadow === bits ) begin
         if ( expect_rdy_0 ) begin
            n_cmd[cmd_given]++;
            n_cyc[cmd_given] += n_cycles;
            lat_range[cmd_given][pos_given].lat_cnt[n_cycles]++;
            pos_range[cmd_given][n_cycles].lat_cnt[pos_given]++;
         end
      end else begin
         n_err++;
         if ( n_err < 5 ) begin
            $write("%s",history_trace);
            $write("Error in %-35s: %h != %h (correct)\n",
                   cmd_info, bits, bits_shadow);
         end
         history_trace = "";
         bits_shadow = bits;
      end

      if ( cmd_given == Cmd_Reset ) history_trace = "";

      history_trace =
        { history_trace,
          $sformatf("Cycle %3d -- %-35s: bits = %h\n",
                    cycle, cmd_info, bits) };

   end

   $write("Finished %0d tests for (wb=%0d,wi=%0d), %0d data errors.\n",
          n_tests, bsize, isize, n_err );

   begin
      automatic bit double_check = 0;
      automatic Command mcc[] = '{ Cmd_Rot_To, Cmd_Write };
      automatic string err_str =
        $sformatf("Error: (wb=%0d,wi=%0d)",bsize,isize);
      $write("Avg cyc");
```

```
        foreach ( mcc[i] )
          $write("  %s %.1f (%0d)",
                 cmd_str[mcc[i]],
                 n_cmd[mcc[i]] ? real'(n_cyc[mcc[i]])/n_cmd[mcc[i]] : 0.0,
                 n_cmd[mcc[i]]);
        $write("\n");
        if ( double_check ) begin
        $write("Avg cyc");
        foreach ( mcc[i] ) begin
           automatic Command c = mcc[i];
           automatic int tot_cyc = 0, tot_cmd = 0;
           foreach ( lat_range[c][pos] ) begin
              foreach ( lat_range[c][pos].lat_cnt[nc] ) begin
                 automatic int ncmd = lat_range[c][pos].lat_cnt[nc];
                 tot_cyc += nc * ncmd;
                 tot_cmd += ncmd;
              end
           end
           $write("  %s %.1f (%0d)",
                  cmd_str[mcc[i]],
                  real'(tot_cyc)/tot_cmd, tot_cmd);
        end
        end
        $write("\n");
        foreach ( mcc[i] ) begin
           automatic Command c = mcc[i];
           automatic int n_one = 0, n_zero = 0;
           string n_z_str, n_o_str;
           n_o_str = $sformatf(" %s 1-cyc pos ",cmd_str[c]);
           foreach ( pos_range[c][1].lat_cnt[pos] ) begin
              n_o_str = { n_o_str, $sformatf("%0d ",pos) };
              n_one++;
           end
           n_z_str = $sformatf(" %s 1-cyc pos ",cmd_str[c]);
           foreach ( pos_range[c][0].lat_cnt[pos] ) begin
              n_z_str = { n_z_str, $sformatf("%0d ",pos) };
              n_zero++;
           end
           if ( n_one ) $write("%s\n",n_o_str);
           if ( n_zero )
             $write("%s\n%s Zero-Cycle %s. Should never be zero when pos!=0\n",
                    n_z_str,err_str,cmd_str[c]);
           if ( c == Cmd_Rot_To && n_one > 2 )
             $write("%s One-Cycle Cmd_Rot_To for more than 2 pos values.\n",
                    err_str);
           if ( c == Cmd_Write && n_one > 0 )
             $write("%s One-Cycle Cmd_Write at least one time. Should never happen.\n",err_str);
        end
      end
      testbench.t_errs += n_err;
      done = 1;

   end

endmodule

// cadence translate_on
```

LSU EE 4755 Homework 5 Solution Due: 17 November 2021

Problem 1: Solve 2020 Solve-Home Final Exam Problem 1, which asks for the inferred hardware for the $v_0^2 + v_0v_1 + v_1^2$ module that we covered in class. For those who may have forgotten how to use a pencil, or never learned, an SVG version of the illustration is available at `https://www.ece.lsu.edu/koppel/v/2020/fe-ms.svg`. Use Inkscape or your favorite SVG editor on the file.

See the 2020 Final Exam Solution.

Problem 2: This assignment does not have a Problem 2. I know that's confusing but the alternative is also confusing.

Problem 3: Solve 2020 Solve-Home Final Exam Problem 3, which asks for a timing analysis of the $v_0^2 + v_0v_1 + v_1^2$ module. An SVG version of the diagram is at `https://www.ece.lsu.edu/koppel/v/2020/fe-ms-t.svg`.

See the 2020 Final Exam Solution for this problem too.

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2021 Homework 6
//
 /// SOLUTION

 /// Assignment https://www.ece.lsu.edu/koppel/v/2021/hw06.pdf


 /// Additional Resources
  //
  // Verilog Documentation
  //    The Verilog Standard
  //      https://ieeexplore.ieee.org/document/8299595/
  //    Introductory Treatment (Warning: Does not include SystemVerilog)
  //      Brown & Vranesic, Fundamentals of Digital Logic with Verilog, 3rd Ed.
  //
  // Account Setup and Emacs (Text Editor) Instructions
  //      https://www.ece.lsu.edu/koppel/v/proc.html
  //      To learn Emacs look for Emacs tutorial.
  //


`default_nettype none


////////////////////////////////////////////////////////////////////////////////
/// Problem 1
//
 ///  Complete multi_step_pipe so that it computes the same value as
 ///  ms_functional, but does so in a pipelined fashion.
 ///
//
//     [✔] Only modify module multi_step_pipe.
//     [✔] Module must operate in pipelined fashion ..
//     [✔] .. meaning it should accept a new set of inputs each cycle ..
//     [✔] .. and provide the result several cycles later.
//     [✔] Be sure to pass the start signal from input to output.
//
//     [✔] Instantiate as many Chipware mult and add units as needed.
//     [✔] The critical path can go through at most one Chipware module.
//
//     [✔] Use SimVision to debug. Use command: xrun -gui hw06.v
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//
//     [✔] Code must be written clearly.
//     [✔] Pay attention to cost and performance.
//
//     [ ] Students can work in teams. List team members in this file

module multi_step_pipe
  ( output logic [31:0] result,
    output logic ready,
    input uwire [31:0] v0, v1,
    input uwire start, clk);

   /// SOLUTION
   //
   //  Part of the solution is changing the object kind of the result
   //  and ready outputs from uwire to var (logic).

   localparam int nstages = 3;

   localparam logic [2:0] rm = 0; // Rounding Mode

   // Wires for fp unit outputs.
   //
   uwire [7:0] mul_s1, mul_s2, mul_s3, a_s1, a_s2;
   uwire [31:0] v00, v01, v11, s1, s2;

   /// Pipeline Latch Declarations
   //
   logic [31:0] pl_1_v00, pl_1_v01, pl_1_v11;
   logic [31:0] pl_2_v0001, pl_2_v11;
   logic pl_1_occ, pl_2_occ;
   //
   // By convention pipeline latch names start with "pl_" followed by
   // the stage in which their value is used (read). So pl_1_v00 holds
```

```
   // a value that will be used in stage 1. The value of pl_1_v00
   // would have to have been written in stage 0.

   /// Floating-Point Functional Unit Instantiations
   //
   //  Instantiate one functional unit for each operation:
   //    v0 * v0  +  v0 * v1  +  v1 * v1
   //    Three multiplications, two additions.
   //
   //  The multipliers' operands come directly from the module inputs ..
   //  .. and so the multipliers are in stage 0.
   //
   CW_fp_mult m00( .z(v00), .a(v0), .b(v0), .rnd(rm), .status(mul_s1) );
   CW_fp_mult m01( .z(v01), .a(v0), .b(v1), .rnd(rm), .status(mul_s2) );
   CW_fp_mult m11( .z(v11), .a(v1), .b(v1), .rnd(rm), .status(mul_s3) );
   //
   //  The adders' operands come from the pipeline latches.
   //
   //  Adder a1 is in stage 1.
   CW_fp_add  a1( .z(s1), .a(pl_1_v00),   .b(pl_1_v01), .rnd(rm), .status(a_s1));
   //
   //  Adder a2 is in stage 2.
   CW_fp_add  a2( .z(s2), .a(pl_2_v0001), .b(pl_2_v11), .rnd(rm), .status(a_s2));

   always_ff @( posedge clk ) begin

      /// Stage 0
      //
      // Stage 0 computes:
      //
      //   v00 <- v0 * v0  (Instance m00)
      //   v01 <- v0 * v1  (Instance m01)
      //   v11 <- v1 * v1  (Instance m11)
      //
      // Write values from stage 0 into pipeline latches.
      //
      pl_1_v00 <= v00;
      pl_1_v01 <= v01;
      pl_1_v11 <= v11;
      pl_1_occ <= start;  // Note that start is passed down pipeline.

      /// Stage 1
      //
      // Stage 1 computes: s1 <- pl_1_v00 + pl_1_v01
      //
      pl_2_v0001 <= s1;
      pl_2_v11 <= pl_1_v11;
      pl_2_occ <= pl_1_occ;

      /// Stage 2
      //
      //  Stage 2 computes: s2 <- pl_2_v0001 + pl_2_v11
      //
      result <= s2;
      ready <= pl_2_occ;
      //
      // Note: result and ready could have been named pl_3_result and
      // pl_3_ready.

   end

endmodule


 /// Non-Synthesizable Mag Module --- Complete, Don't Edit
// cadence translate_off
module multi_step_functional
  ( output shortreal mag,
    input shortreal v0, v1 );

   always_comb mag = v0 * v0 + v0 * v1 + v1 * v1;

endmodule
// cadence translate_on



 /// Non-Synthesizable Mag Module --- Complete, Don't Edit
//
// This is provided for reference.
//
```

```
module multi_step_seq
  ( output logic [31:0] result,
    output logic ready,
    input uwire [31:0] v0, v1,
    input uwire start,
    input uwire clk );

   localparam logic [2:0] rnd = 0; // 1 is round towards zero.

   uwire [7:0] mul_s, add_s;

   logic [2:0]  step;

   uwire [31:0] mul_a, mul_b;
   uwire [31:0] add_a, add_b;
   uwire [31:0] prod, sum;

   logic [31:0] ac0, ac1;

   localparam   int last_step = 4;

   always_ff @( posedge clk )
     if ( start ) step <= 0;
     else if ( step < last_step ) step <= step + 1;

   CW_fp_mult m1( .a(mul_a), .b(mul_b), .rnd(rnd), .z(prod), .status(mul_s));
   CW_fp_add  a1( .a(add_a), .b(add_b), .rnd(rnd), .z(sum),  .status(add_s));

   assign mul_a = step < 2  ? v0 : v1;
   assign mul_b = step == 0 ? v0 : v1;
   assign add_a = ac0,  add_b = ac1;

   always_ff @( posedge clk )
     begin

        ac0 <= prod;

        case ( step )
           0: ac1 <= 0;
           1: ac1 <= sum;
           2: ac1 <= sum;
        endcase

        if ( start ) ready <= 0; else if ( step == last_step-1 ) ready <= 1;

     end

   assign result = sum;

endmodule




////////////////////////////////////////////////////////////////////////////////
/// Testbench Code


// cadence translate_off

function automatic real rand_real(real minv, real maxv);
      rand_real = minv + ( maxv - minv ) * ( real'({$random}) ) / 2.0**32;
endfunction

function automatic shortreal fabs(shortreal val);
      fabs = val < 0 ? -val : val;
endfunction

program reactivate
   (output uwire clk_reactive, output int cycle_reactive,
    input uwire clk, input var int cycle);
   assign clk_reactive = clk;
   assign cycle_reactive = cycle;
endprogram

module testbench;

   typedef enum { MT_comb, MT_seq, MT_pipe } Module_Type;

   localparam int num_tests = 400;
   localparam int NUM_MUT = 4;
```

```
localparam int err_limit = 7;
localparam int trace_max_lines = 10;

shortreal magr, vr[2];
logic [31:0] vp[NUM_MUT][2];
uwire [31:0] mag[NUM_MUT];

uwire availn[NUM_MUT];
logic avail[NUM_MUT];
logic in_valid[NUM_MUT];

typedef struct { int tidx; int cycle_start; int eta; } Test_Vector;

typedef struct { int idx;
                 int err_count = 0;
                 int err_timing = 0;
                 Module_Type mt = MT_comb;
                 Test_Vector tests_active[$];
                 string trace_lines[$];
                 int eta_to_test[int];
                 bit all_tests_started = 0;
                 bit seq = 0; bit pipe = 0;
                 bit bpipe = 0;
                 int ncyc = 0;
                 int ncompleted = 0;
                 int cyc_tot = 0;
                 int latency = 0;
                 } Info;
Info pi[string];

localparam int cycle_limit = num_tests * 10;
int cycle;
bit done;
logic clock;

logic clk_reactive;
int cycle_reactive;
reactivate ra(clk_reactive,cycle_reactive,clock,cycle);

initial begin
   clock = 0;
   cycle = 0;

   fork
      forever #10 cycle += ++clock;
      wait( done );
      wait( cycle >= cycle_limit )
        $write("*** Cycle limit exceeded, ending.\n");
   join_any;

   $finish();
end

task pi_seq(input int idx, input string name);
   automatic string m = $sformatf("%s", name);
   pi[m].idx = idx; pi[m].mt = MT_seq;
   pi[m].seq = 1; pi[m].bpipe = 0; pi[m].pipe = 0;
endtask

task pi_pipe(input int idx, input string name, input int ncyc);
   automatic string m = $sformatf("%s", name);
   pi[m].idx = idx; pi[m].mt = MT_pipe;
   pi[m].ncyc = ncyc;
   pi[m].seq = 1; pi[m].pipe = 1; pi[m].bpipe = 0;
endtask


multi_step_pipe m3( mag[3], availn[3], vp[3][0],vp[3][1], in_valid[3], clock );
initial begin pi_pipe(3,"MS Pipe",m3.nstages); end

always @*
  foreach ( availn[i] ) if ( availn[i] !== 1'bz ) avail[i] = availn[i];

initial begin

   automatic int awaiting = pi.size();

   logic [31:0] vs[num_tests][2];
   shortreal vrs[num_tests][2];

   done = 0;
```

```
   foreach ( pi[mut] ) begin
      automatic int midx = pi[mut].idx;
      automatic int steps = pi[mut].ncyc;
      automatic int latency =
        !pi[mut].seq ? 1 : !pi[mut].pipe ? 2 * steps : steps;
      pi[mut].latency = latency;
      in_valid[midx] = 0;
   end

   for ( int i=0; i<num_tests; i++ ) begin

      if ( i < 4 ) begin

         // In first eight tests vector components are zero or one.
         //
         for ( int j=0; j<2; j++ ) vrs[i][j] = i & 1 << j ? 1.0 : 0.0;

      end else begin

         // In other tests vector components are randomly chosen.
         //
         for ( int j=0; j<2; j++ ) vrs[i][j] = rand_real(-10,+10);

      end

      for ( int j=0; j<2; j++ ) vs[i][j] = $shortrealtobits(vrs[i][j]);

   end

   fork forever @( negedge clk_reactive ) foreach ( pi[mut] ) begin
      automatic int midx = pi[mut].idx;
      if ( !in_valid[midx] && pi[mut].pipe ) begin
         vp[midx][0] = cycle;
         vp[midx][1] = 1;
      end
   end join_none;

   repeat ( 2 * 10 ) @( negedge clock );

   foreach ( pi[mutii] ) begin
      automatic string muti = mutii;

      fork begin
         automatic string mut = muti;
         automatic int midx = pi[mut].idx;
         for ( int i=0; i<num_tests; i++ ) begin
            automatic int gap_cyc =
              ( {$random} % 2 ) ? {$random} % ( 5 ) : 0;
            automatic Test_Vector tv;
            repeat ( gap_cyc ) @( negedge clock );
            vp[midx] = vs[i];
            in_valid[midx] = 1;
            tv.tidx = i;
            tv.cycle_start = cycle;
            tv.eta = tv.cycle_start + pi[mut].latency;
            pi[mut].eta_to_test[tv.eta] = i;
            pi[mut].tests_active.push_back( tv );
            @( negedge clock );
            in_valid[midx] = 0;
         end
         pi[mut].all_tests_started = 1;
      end join_none;

      fork begin
         automatic string mut = muti;
         automatic int midx = pi[mut].idx;
         automatic int n_timing_errs = 0;
         automatic int n_correct_val = 0; // Reset when test over.
         while ( !pi[mut].all_tests_started || pi[mut].tests_active.size() )
           @( negedge clk_reactive ) begin
              automatic shortreal v0 = $bitstoshortreal(vp[midx][0]);
              automatic shortreal v1 = $bitstoshortreal(vp[midx][1]);
              automatic shortreal r_future = v0*v0+v0*v1+v1*v1;
              automatic shortreal r = $bitstoshortreal(mag[midx]);
              automatic Test_Vector tv = pi[mut].tests_active[0];
              automatic bit avail_sh = pi[mut].eta_to_test.exists(cycle);
              automatic int ita = tv.tidx;
              automatic int ieta =
                avail_sh ? pi[mut].eta_to_test[cycle] : -1;
              automatic int i = ita > ieta ? ita : ieta;
```

```
                automatic shortreal v0p = vrs[i][0], v1p = vrs[i][1];
                automatic shortreal shadow_magr = v0p*v0p+v0p*v1p+v1p*v1p;
                automatic string in_txt = in_valid[midx]
                  ? $sformatf("In: %5.1f,%5.1f -> %5.1f", v0, v1, r_future)
                  : "start=0";
                automatic shortreal err_mag = fabs( r - shadow_magr );
                automatic bit okay = err_mag < 1e-4;
                automatic bit err_rdy = avail_sh !== avail[midx];
                automatic bit err_val = avail_sh && !okay;
                automatic string tr_txt =
                  $sformatf
                    ("%-8s Cyc %3d  %-24s  Rdy %1d%s, Res: %5.1f %0s\n",
                     mut, cycle, in_txt,
                     avail[midx],
                     err_rdy ? "X" : " ",
                     r,
                     okay &&  avail[midx] && avail_sh  ? "Good" :
                     okay && !avail[midx] && avail_sh  ? "XX: Need Rdy" :
                     okay && avail[midx] && !avail_sh ? "XX: Early" :
                     !okay && avail_sh ? "XX: Wrong" :
                     avail[midx] && !avail_sh ? "XX: Unexpected" : ""
                     );

                if ( err_rdy ) n_timing_errs++;
                if ( okay ) n_correct_val++;

                if ( pi[mut].ncompleted < 3 )
                  $write("%s",tr_txt);
                else
                  pi[mut].trace_lines.push_back( tr_txt );

                if ( pi[mut].err_count < err_limit
                     && pi[mut].err_timing < err_limit
                     && ( err_rdy || err_val ) )
                  while ( pi[mut].trace_lines.size() )
                    $write("%s", pi[mut].trace_lines.pop_front() );

                if ( avail_sh ) begin
                   pi[mut].tests_active.delete(0);
                   pi[mut].ncompleted++;

                   if ( n_timing_errs ) begin
                      pi[mut].err_timing++;
                      n_timing_errs = 0;
                   end

                   if ( n_correct_val == 0 ) begin
                      pi[mut].err_count++;
                      if ( pi[mut].err_count <= err_limit ) begin
                         $write
                           ("%-8s test %0d: Inputs at cyc %0d, result expected at cyc %0d. Wrong val: h'%8h  %7.4f != %7.4f (correct)\n",
                            mut, i, tv.cycle_start, tv.eta,
                            mag[midx], r, shadow_magr);
                      end
                   end
                   n_correct_val = 0;
                end

                if ( pi[mut].trace_lines.size() > trace_max_lines )
                  pi[mut].trace_lines.delete(0);
              end
            awaiting--;
          end join_none
        end

        wait( awaiting == 0 || cycle > cycle_limit );

        foreach ( pi[ mut ] )
          $write("For %0s ran %0d tests: Errors: %0d wrong val, %0d bad timing\n",
                 mut, num_tests,
                 pi[mut].err_count, pi[mut].err_timing);


        done = 1;

        $finish(2);

      end

    endmodule
```

```
// cadence translate_on

`default_nettype wire

`include "/apps/linux/cadence/GENUS211/share/synth/lib/chipware/sim/verilog/CW/CW_fp_mult.v"
`include "/apps/linux/cadence/GENUS211/share/synth/lib/chipware/sim/verilog/CW/CW_fp_add.v"
```

18 Fall 2020 Solutions

LSU EE 4755 Homework 1 Solution Due: 16 September 2020

☣ *Paper copies will not be accepted. E-mail your solution to* `koppel@ece.lsu.edu`. *A single PDF file is preferred.*

Problem 1: In the Module-Port-versus-Module-Parameter section of lecture code `https://www.ece.lsu.edu/koppel/v/2020/l005-review.v.html` there are several module designs for computing $c_1x + c_2y$, where c_1 and c_2 are constants and x and y are module inputs. The point of that section and of the modules was to illustrate the SystemVerilog differences between module parameters and ports (syntax issues, for example) and also how they relate to the hardware being modeled.

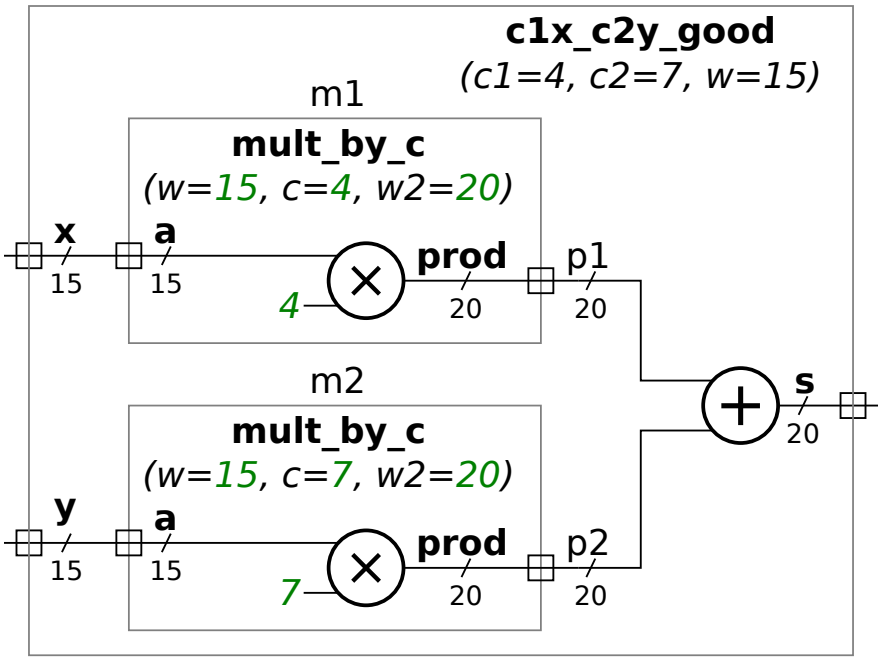
(*a*) Draw a diagram of module `c1x_c2y_good`, shown below, using its default parameter values (which are different than the ones in the lecture code). Show the contents of all instantiated modules and appropriately label ports and wires. (See 2016 Homework 1 Problem 3 for a diagram showing instantiated modules. Also see module `arb_exp` and the illustration that follows in `https://www.ece.lsu.edu/koppel/v/2020/l015-syn-comb-str.v.html`.)

- Use the default parameter values of the module `c1x_c2y_good` shown below.
- Use the appropriate parameter values for the `mult_by_c` instances. *Hint: appropriate is not a synonym for default.*
- Show the ports for all modules.
- Show the number of bits in each wire.
- Label wires with the symbols used below (such as `p1` and `prod`) and take care to place the label on the correct side of a module boundary. (In the `two_pie` illustration from `https://www.ece.lsu.edu/koppel/v/2020/l005-review.v.html` look at the wire carrying labels `x`, `i1`, and `a`.).

Continued on next page.

```
module mult_by_c
  #( int w = 8, int c = 16, int w2 = w+$clog2(c) )
   ( output uwire signed [w2-1:0] prod, input uwire signed [w-1:0] a );
   assign prod = a * c;
endmodule

module c1x_c2y_good
  #( int c1 = 4, int c2 = 7, int w = 15,
     int w2 = w + $clog2(c1) + $clog2(c2) )
   ( output logic signed [w2-1:0] s,   input uwire signed [w-1:0] x, y );

   uwire [w2-1:0] p1, p2;

   mult_by_c #(w,c1,w2) m1(p1,x);
   mult_by_c #(w,c2,w2) m2(p2,y);

   assign  s = p1 + p2;

endmodule
```

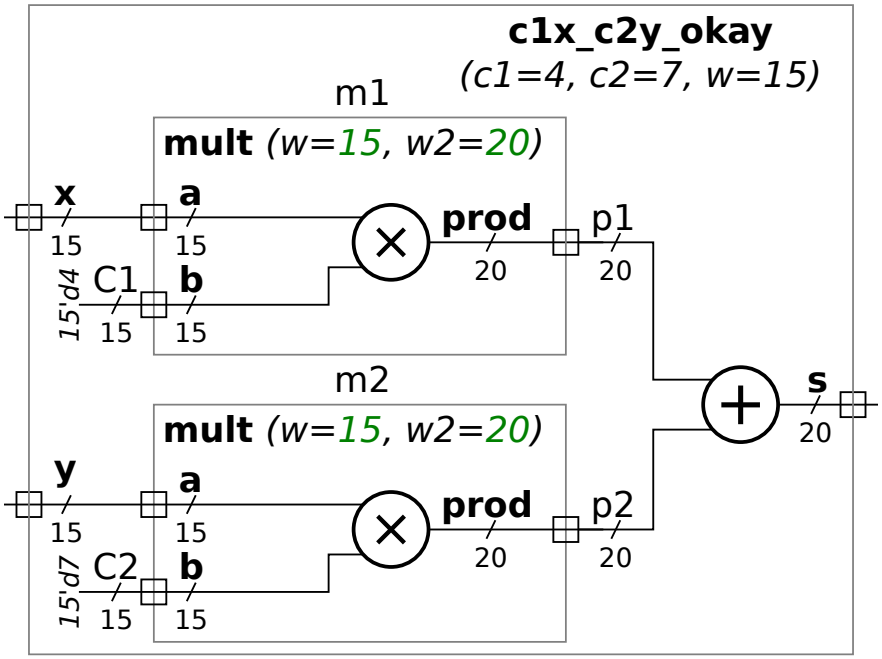
Solution appears below. Notice that parameters are not shown as module inputs. For example, `c1` is not shown as an input to `m1`.



(*b*) Draw a diagram of module `c1x_c2y_okay` below using its default parameter values (which are different than the defaults used in the lecture code). Show the same details, such as ports, as was requested for the previous part.

```
module mult
  #( int w = 8, int w2 = 2 * w )
   ( output uwire signed [w2-1:0] prod, input uwire signed [w-1:0] a, b );
   assign prod = a * b;
endmodule

module c1x_c2y_okay
  #( int c1 = 4, int c2 = 7, int w = 15,
     int w2 = w + $clog2(c1) + $clog2(c2) )
   ( output logic signed [w2-1:0] s,   input uwire signed [w-1:0] x, y );

   uwire [w2-1:0] p1, p2;
   uwire [w:1] C1 = c1, C2 = c2;  // Convert constants to desired size.

   mult #(w,w2) m1(p1, x, C1);
   mult #(w,w2) m2(p2, y, C2);

   assign  s = p1 + p2;

endmodule
```

Solution appears below. Here, `C1` and `C2` are inputs to `m1` and `m2`. A lazy synthesis program, or less judgmentally, a synthesis program set to optimize at a low effort level might not take advantage of the fact that in `m1` the `b` input is 4. That would result in much more expensive hardware.

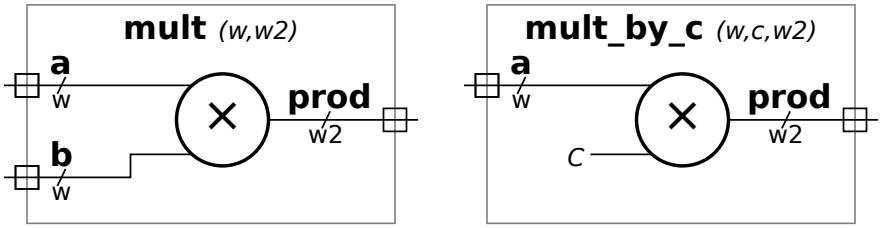


Problem 2: Synthesis programs optimize a design to minimize cost while meeting timing constraints. The illustration below for the `mult` and `mult_by_c` modules (used in the slides) show how the multiplier can be simplified when one of the inputs is a convenient constant, 1.

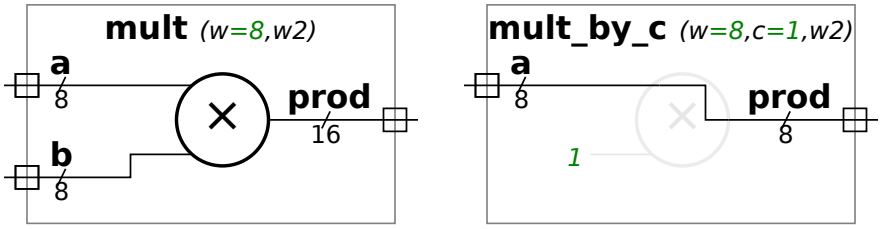
Show how the `c1x_c2y_good` module from the first problem can be optimized based on the default `c1=4` and `c2=7` values. To do so show the multiplier replaced by much simpler hardware, such as adder(s). A correct solution uses only one adder for both multipliers, bit relabeling, *plus the adder used to combine* `p1` *and* `p2`.

Note: As originally assigned, and until Tuesday, 15 September 2020 at about 16:15, the problem stated that a correct solution uses only one adder, implying but not specifically stating that the one adder was the replacement for the multipliers and that there would also be and adder computing `p1+p2`, for a total of two adders.

Before instantiation and optimization.



After instantiation and optimization.



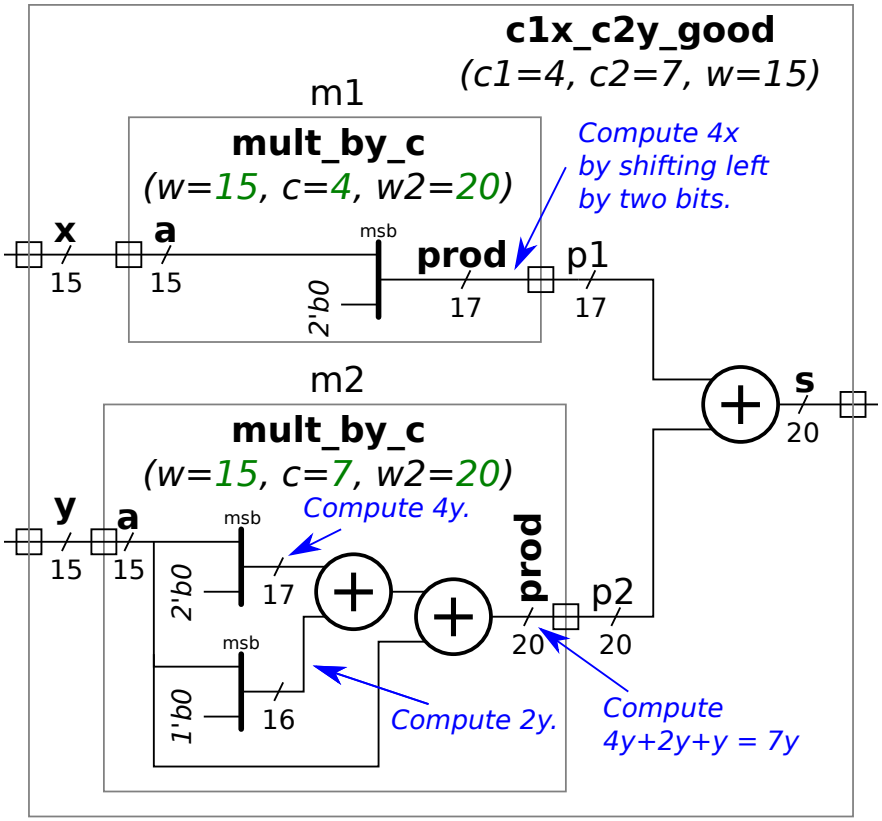
Solution on next page.

Two solutions appear below. The first is easier to understand, but uses two adders for `m2`. The second uses one adder for `m2`.

Both solutions take advantage of the fact that multiplication by a power of 2, such as 4, can be achieved by left-shifting. To compute $4x$ the value of x is left-shifted by two positions. The hardware for achieving that is trivial: relabel bit position i to $i+2$ and set bits at positions 0 and 1 to the constant 0. Both solutions do this in `m1`. Make sure that the notation for re-labeling bits used in `m1` is understood.

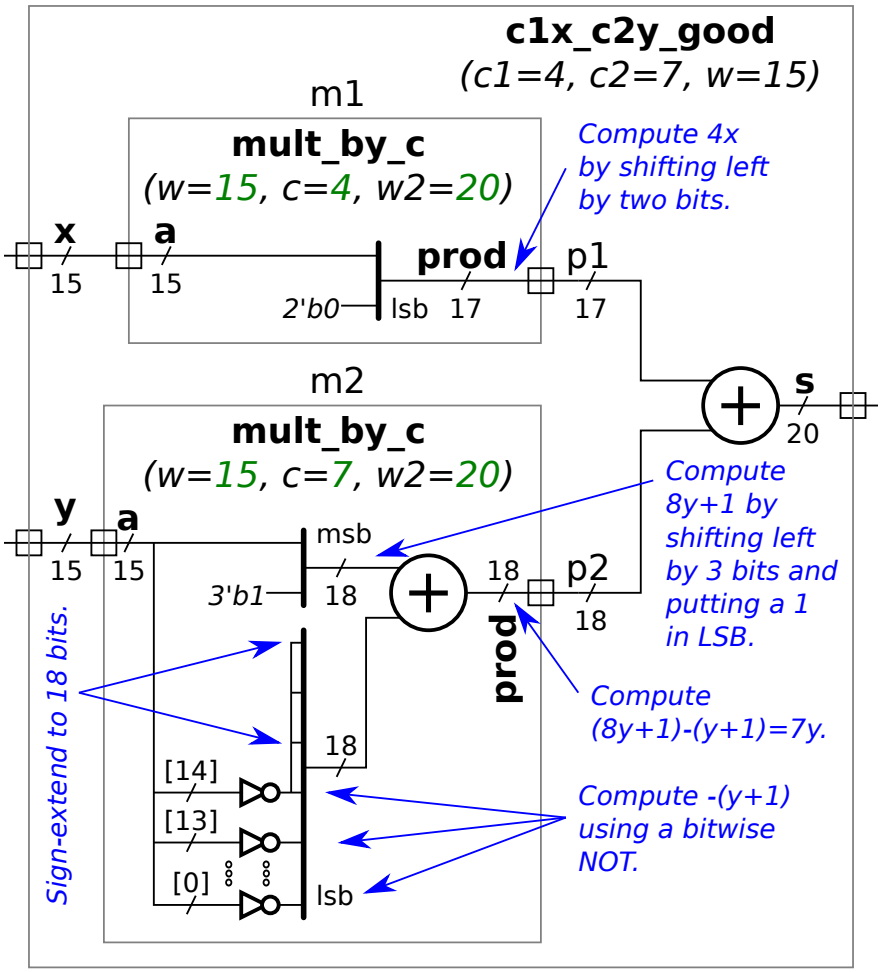
The solution below computes $7y$ using two adders: $4y + 2y + y = 7y$.

Both solutions use adders that have unequal port sizes. For example in the first solution the adder computing `s` has one 17-bit input and one 20-bit input. That's not an unreasonable assumption to make.



Better one-adder-`m2` solution on next page.

The solution using one adder for `m2` appears below. Recall that `m2` computes $7y$. That can be done with one adder by computing $8y + (-y) = 7y$. But to compute a 2's complement representation of $-y$ one needs to negate each bit and then add 1. Negating each bit is easy. A wasteful solution would use an adder just to compute $(-y-1)+1$. There's no need for that here, instead the solution computes $(8y+1)+(-y-1) = 7y$. The quantity $8y+1$ is obtained by left-shifting by 3 bits and then putting a 1 in the least-significant bit position. Negating the bits of 2's complement number y results in $-y-1$, which is what we need. Notice that the hardware computing $-y-1$ produces an 18-bit quantity by sign-extending the 15-bit quantity. The need to do sign extension in the diagram below could have been eliminated by using an adder with an 18- and 15-bit input. The adder would do the sign-extension internally.



```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2020 Homework 2 -- SOLUTION
//
 /// Assignment https://www.ece.lsu.edu/koppel/v/2020/hw02.pdf



`default_nettype none

////////////////////////////////////////////////////////////////////////////////
/// Problem 1
//
 /// Modify
 ///
//
//     [✔] nn4x4b must instantiate exactly four nn1x4b modules.
//     [✔] nn1x4b must instantiate exactly two nn1x2 modules.
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//
//     [✔] Don't assume any particular parameter value.
//     [✔] Pay attention to port widths. Do not make them larger than needed.
//
//     [✔] Code must be written clearly.

module nn4x4
  #( int wa = 10, ww = 5 )
   ( output uwire [wa-1:0] ao[4],
     input uwire [wa-1:0] ai[4],
     input uwire [ww-1:0] wht[4][4] );

   /// DO NOT MODIFY THIS ROUTINE.

   assign ao[0] = ai[0] * wht[0][0] + ai[1] * wht[0][1]
          + ai[2] * wht[0][2] + ai[3] * wht[0][3];

   assign ao[1] = ai[0] * wht[1][0] + ai[1] * wht[1][1]
          + ai[2] * wht[1][2] + ai[3] * wht[1][3];

   assign ao[2] = ai[0] * wht[2][0] + ai[1] * wht[2][1]
          + ai[2] * wht[2][2] + ai[3] * wht[2][3];

   assign ao[3] = ai[0] * wht[3][0] + ai[1] * wht[3][1]
          + ai[2] * wht[3][2] + ai[3] * wht[3][3];

endmodule

module nn4x4b
  #( int wa = 10, ww = 5 )
```

```
   ( output uwire [wa-1:0] ao[4],
     input uwire [wa-1:0] ai[4],
     input uwire [ww-1:0] wht[4][4] );

   /// SOLUTION

   nn1x4b #(wa,ww) n0( ao[0], ai, wht[0] );
   nn1x4b #(wa,ww) n1( ao[1], ai, wht[1] );
   nn1x4b #(wa,ww) n2( ao[2], ai, wht[2] );
   nn1x4b #(wa,ww) n3( ao[3], ai, wht[3] );

endmodule

module nn1x4b
  #( int wa = 10, ww = 5 )
   ( output uwire [wa-1:0] ao,
     input uwire [wa-1:0] ai[4],
     input uwire [ww-1:0] wht[4] );

   /// SOLUTION

   uwire [wa-1:0] aoa, aob;
   nn1x2b #(wa,ww) n0(aoa, ai[0:1], wht[0:1] );
   nn1x2b #(wa,ww) n1(aob, ai[2:3], wht[2:3] );
   assign ao = aoa + aob;

endmodule

module nn1x2b
  #( int wa = 10, ww = 5 )
   ( output uwire [wa-1:0] ao,
     input uwire [wa-1:0] ai[2],
     input uwire [ww-1:0] wht[2] );

   /// SOLUTION

   assign ao = ai[0] * wht[0] + ai[1] * wht[1];

endmodule



////////////////////////////////////////////////////////////////////////////////
/// Testbench Code

module nn0xl
  #( int no = 4, ni = 4, wa = 10, ww = 5 )
   ( output logic [wa-1:0] ao[no],
     input uwire [wa-1:0] ai[ni],
     input uwire [ww-1:0] wht[no][ni] );

      /// DO NOT MODIFY THIS ROUTINE.

     always_comb
```

```
      for ( int o = 0;  o < no;  o++ ) begin
         ao[o] = 0;
         for ( int i=0; i<ni; i++ ) ao[o] += ai[i] * wht[o][i];
      end

endmodule

// cadence translate_off

module testbench;

   localparam int wa = 16;
   localparam int ww = 8;
   localparam int ni = 4;  // Number of input neurons.
   localparam int no = 4;  // Number of output neurons.
   localparam int nmut = 3;

   localparam int ntests = 10;

   logic [wa-1:0] ai[ni];
   uwire [wa-1:0] ao[nmut][no];
   logic [ww-1:0] wht[no][ni];

   string mname[] = { "Behav", "Flat", "Sol" };

   typedef struct { string name; int no, ni; } Test_Set;
   Test_Set ts[] = '{ '{ "n12", 1, 2 }, '{ "n14", 1, 4 }, '{ "n44", 4, 4 } };

   nn0xI #(no,ni,wa,ww) nn1i(ao[0],ai,wht);
   nn4x4 #(wa,ww) nn2i(ao[1],ai,wht);
   nn4x4b #(wa,ww) nn3i(ao[2],ai,wht);

   initial begin

      automatic int mut = 2;
      automatic string test_summary = "";

      $write("Testing module %s\n", mname[mut]);

      foreach ( ts[ti] ) begin

         automatic Test_Set tinfo = ts[ti];
         automatic int n_err = 0;

         $write("\n** Starting test set %s  (%0d outputs, %0d inputs) **\n",
                tinfo.name, tinfo.no, tinfo.ni );

         for ( int tnum=0; tnum < ntests;  tnum++ ) begin

            for ( int io=0; io<no; io++ )
              for ( int ii=0; ii<ni; ii++ )
                wht[io][ii] = io < tinfo.no && ii < tinfo.ni ? {$random} : 0;

            for ( int ii=0; ii<ni; ii++ )
```

```
            ai[ii] = ii < tinfo.ni ? {$random} : 0;

         #1;

         for ( int io=0; io<tinfo.no; io++ ) begin

            if ( ao[0][io] !== ao[mut][io] ) begin
               n_err++;
               if ( n_err < 4 )
                 $write
                   ("Error test # %0d, output %0d: %0d != %0d (correct)\n",
                    tnum, io, ao[mut][io], ao[0][io] );
            end
         end
      end

      begin
         automatic string msg =
           $sformatf("%0d %s tests on %s: %0d errors found.",
                     ntests, tinfo.name, mname[mut], n_err);
         $write("Done with %s\n",msg);
         test_summary = { test_summary, $sformatf("Results of %s\n", msg) };
      end

   end

   $write("\n** Summary of Results **\n%s", test_summary);

end


endmodule

// cadence translate_on
```

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2020 Homework 3 -- SOLUTION
//

 /// Assignment https://www.ece.lsu.edu/koppel/v/2020/hw03.pdf


`default_nettype none

////////////////////////////////////////////////////////////////////////////////
/// Problem 1
//
 /// Modify nnOxI and nn1xI so they compute same output as nnOxIbe.
 ///
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//
//     [✔] Don't assume any particular parameter value.
//     [✔] Don't make ports wider than necessary.
//
//     [✔] Code must be written clearly.


   /// Module Connection Names
   //
   //  no:  Number of elements in Output array.
   //  ni:  Number of elements in Input array.
   //  wo:  Width (number of bits) in each element of output array.
   //  wi:  Width (number of bits) in each element of input array.
   //  ww:  Width (number of bits) in each element of weight array.
   //  sat: If 0, on overflow use low wo bits of result.
   //       If 1, on overflow set result to maximum possible value,
   //             do this where overflow occurs.
   //       If 2, on overflow set result to maximum possible value,
   //             do this in nnOxI (not in nn1xI nor in nnAdd, nnMult, etc.)
   //  tr:  If 0, generate a linear connection of nnMADD modules in nn1xI.
   //       If 1, generate a tree connection of arithmetic units by
   //             recursively defining nn1xI.
   //  ao:  Activation (neuron) Output array.
   //  ai:  Activation (neuron) Input array.
   //  wht: Weights.


module nnOxI
  #( int no = 4, ni = 2, wo = 10, wi = 4, ww = 5, tr = 0, sat = 0 )
   ( output uwire [wo-1:0] ao[no],
     input uwire [wi-1:0] ai[ni],
     input uwire [ww-1:0] wht[no][ni] );

   //   [✔] Instantiate nn1xI modules here.
   //   [✔] If sat == 2 replace overflow values with max possible value.
```

```
   //   [✔] Don't forget to set appropriate parameter values.

   /// SOLUTION

   // Compute number of bits to represent largest possible value that
   // can appear on an ao.
   //
   localparam int wr = $clog2( ( 2**wi - 1 ) * ( 2**ww  - 1 ) * ni );

   if ( sat < 2 || wr <= wo ) begin

      // If overflow is not possible turn off check for saturation.
      //
      localparam int satp = wr <= wo ? 0 : sat;

      for ( genvar i = 0;  i < no;  i++ )
        nn1xI #(wo,wi,ww,ni,tr,satp) row( ao[i], ai, wht[i] );

   end else begin

      for ( genvar i = 0;  i < no;  i++ ) begin

         uwire [wr-1:0] ar;
         nn1xI #(wr,wi,ww,ni,tr,0) row( ar, ai, wht[i] );

         // If there is an overflow substitute maximum value.
         //
         assign ao[i] = ar[wr-1:wo] ? ~wo'(0) : ar[wo-1:0];

      end

   end

endmodule

module nn1xI
  #( int wo = 10, wi = 4, ww = 5, ni = 2, tr = 0, sat = 0 )
   ( output uwire [wo-1:0] ao,
     input uwire [wi-1:0] ai[ni],
     input uwire [ww-1:0] wht[ni] );

   //   [✔] If tr == 0 use generate loop to instantiate nnMADD modules.
   //   [✔] If tr == 1 use recursion to describe a tree structure ..
   //   [✔] .. and use nnMADD, nnMult, and nnAdd where appropriate.
   //   [✔] Don't forget to set appropriate parameter values.


   /// SOLUTION
   //
   if ( tr ) begin

      if ( ni == 1 ) begin

         nnMult #(wi,ww,wo,sat) mult(ao, ai[0], wht[0] );
```

```
      end else begin

         localparam int nlo = ni / 2;
         localparam int nhi = ni - nlo;
         uwire [wo-1:0] aolo, aohi;
         nn1xI #(wo,wi,ww,nlo,1,sat) nnlo(aolo, ai[0:nlo-1], wht[0:nlo-1]);
         nn1xI #(wo,wi,ww,nhi,1,sat) nnhi(aohi, ai[nlo:ni-1], wht[nlo:ni-1]);
         nnAdd #(wo,sat) add(ao,aolo,aohi);

      end

   end else begin

      uwire [wo-1:0] s[ni-1:-1];
      assign s[-1] = 0;
      assign ao = s[ni-1];

      for ( genvar i = 0;  i < ni;  i++ )
        nnMADD #(ww,wi,wo,sat) madd( s[i], wht[i], ai[i], s[i-1] );

   end

endmodule

module nnMADD
  #( int wa = 10, wb = 5, ws = wa + wb, sat = 0 )
   ( output uwire [ws-1:0] so,
     input uwire [wa-1:0] a, input uwire [wb-1:0] b, input uwire [ws-1:0] si);

   /// DO NOT MODIFY THIS MODULE.

   uwire [ws-1:0] p;
   nnMult #(wa,wb,ws,sat) mu(p, a, b);
   nnAdd #(ws,sat) ad(so, si, p);

endmodule

module nnAdd
  #( int w = 5, sat = 0 )
   ( output uwire [w-1:0] so,
     input uwire [w-1:0] a, b );

   /// DO NOT MODIFY THIS MODULE.

   uwire [w:0] s = a + b;
   localparam logic [w-1:0] smax = ~w'(0);
   assign so = sat && s[w] ? smax : s[w-1:0];

endmodule

module nnMult
  #( int wa = 5, wb = 6, wp = wa + wb, sat = 0 )
   ( output uwire [wp-1:0] p,
```

```
    input uwire [wa-1:0] a, input uwire [wb-1:0] b );

  /// DO NOT MODIFY THIS MODULE.

  localparam logic [wp-1:0] pmax = ~wp'(0);
  localparam int wmx = wp > wa+wb ? wp : wa+wb;
  uwire [wmx-wp:0] phi;
  uwire [wp-1:0] plo;
  assign {phi,plo} = a * b;
  assign p = sat && wp < wa + wb && phi ? pmax : plo;

endmodule


// Synthesizing at effort level "medium"

// Module Name                                  Area   Delay   Delay
//                                                    Actual  Target

// nnOxI_no2_ni16_wo12_wi5_ww4_sat0_tr0    588304   6.972  90.000 ns
// nnOxI_no2_ni16_wo12_wi5_ww4_sat0_tr1    588304   6.972  90.000 ns
// nnOxI_no2_ni16_wo12_wi5_ww4_sat1_tr0    753136  63.864  90.000 ns
// nnOxI_no2_ni16_wo12_wi5_ww4_sat1_tr1    631611   7.043  90.000 ns
// nnOxI_no2_ni16_wo12_wi5_ww4_sat2_tr0    594261   7.450  90.000 ns
// nnOxI_no2_ni16_wo12_wi5_ww4_sat2_tr1    594261   7.450  90.000 ns

// nnOxI_no2_ni16_wo12_wi5_ww4_sat0_tr0    783094   4.828   1.000 ns
// nnOxI_no2_ni16_wo12_wi5_ww4_sat0_tr1    779386   4.852   1.000 ns
// nnOxI_no2_ni16_wo12_wi5_ww4_sat1_tr0    951332   9.503   1.000 ns
// nnOxI_no2_ni16_wo12_wi5_ww4_sat1_tr1    916787   5.136   1.000 ns
// nnOxI_no2_ni16_wo12_wi5_ww4_sat2_tr0    800554   4.980   1.000 ns
// nnOxI_no2_ni16_wo12_wi5_ww4_sat2_tr1    771789   4.981   1.000 ns

// Normal exit.



module nnOxIbe
  #( int no = 4, ni = 4, wo = 10, wi = 4, ww = 5, sat = 0 )
   ( output logic [wo-1:0] ao[no],
     input uwire  [wi-1:0] ai[ni], input uwire  [ww-1:0] wht[no][ni] );

  /// DO NOT MODIFY THIS MODULE
  //
  //  Study the code in this module to get a better understanding
  //  of what the output of nnOxI should be.

  // Determine the maximum possible value of each element of ao.
  //
  localparam logic [wo-1:0] smax = ~wo'(0);


  always_comb
    for ( int o = 0;  o < no;  o++ ) begin
```

```
        automatic int unsigned acc = 0;
        for ( int i=0; i<ni; i++ ) acc += ai[i] * wht[o][i];
        ao[o] = sat && acc > smax ? smax : acc;
      end

endmodule

////////////////////////////////////////////////////////////////////////////////
```

/// Testbench Code

```
// cadence translate_off

typedef struct { int ni, no, wa, ww; } Config;

module testbench;

   localparam int nc = 2;
   localparam int configs[nc][4] = '{ '{ 4,4, 16,9 }, '{ 5,3, 15,8 } };
   `ifdef grrr
   initial if ( nc != configs.size() )
     $fatal(1, "Constant nc should be %0d.\n", configs.size() );
   `endif

   int t_errs;     // Total number of errors.
   int t_errs_cat[string];  // Total errors by configuration category.
   string test_summary;
   initial begin
      t_errs = 0;
      test_summary = "";
   end

   final begin

      automatic int mlen = 0;
      foreach ( t_errs_cat[key] ) if ( mlen < key.len() ) mlen = key.len();

      $write("\n** Summary of Results **\n%s", test_summary);
      foreach ( t_errs_cat[key] )
        $write("%0s%0s %5d errors.\n",
               key, { mlen - key.len() {" "}}, t_errs_cat[key]);
      $write("Total number of errors: %0d\n",t_errs);

   end

   localparam int maxsat = 3;

   uwire d[maxsat*nc:-1];    // Start / Done signals.
   assign d[-1] = 1;  // Initialize first at true.

   // Instantiate a testbench at each size.
   //
   for ( genvar i=0; i<nc; i++ ) begin
      localparam int c[4] = configs[i];
```

```
      for ( genvar sat=0; sat<maxsat; sat++ ) begin
         localparam int idx = maxsat*i + sat;
         testbench_x #(c[0],c[1],c[2],c[3],sat)
         t2( .done(d[idx]), .start(d[idx-1]) );
      end
   end

endmodule

module testbench_x
  #( int ni = 4, no = 4, wo = 16, ww = 8, sat = 0 )
   ( output logic done, input uwire start );

   localparam int wi = ww + 1;

   localparam int nmut = 3;

   localparam int ntests = 10;

   logic [wi-1:0] ai[ni];
   uwire [wo-1:0] ao[nmut][no];
   logic [ww-1:0] wht[no][ni];

   string mname[] = { "Behav", "Linear", "Tree" };

   typedef struct { string name; int no, ni; } Test_Set;
   Test_Set ts[] = '{ '{ "n12", 1, 2 }, '{ "n1*", 1, ni }, '{ "n**", no, ni } };

   nnOxIbe #(no,ni,wo,wi,ww,sat) nn0(ao[0],ai,wht);
   nnOxI #(no,ni,wo,wi,ww,0,sat) nn1(ao[1],ai,wht);
   nnOxI #(no,ni,wo,wi,ww,1,sat) nn2(ao[2],ai,wht);

   initial begin

      automatic string config_label =
        $sformatf("no=%0d, ni=%0d, wo=%0d, wi=%0d, ww=%0d, sat=%0d",
                  no, ni, wo, wi, ww, sat );

      wait( start );

      $write("\n** Starting tests for %s\n", config_label);
      testbench.test_summary =
        { testbench.test_summary, $sformatf("Results from %s\n",config_label) };

      for ( int mut = 1;  mut < nmut;  mut++ ) begin

         $write("Testing module %s\n", mname[mut]);

         foreach ( ts[ti] ) begin

            automatic Test_Set tinfo = ts[ti];
            automatic int n_err = 0;

            $write("\n** Starting test set %s  (%0d outputs, %0d inputs) for %s **\n",
```

```
                tinfo.name, tinfo.no, tinfo.ni, mname[mut] );

          for ( int tnum=0; tnum < ntests;  tnum++ ) begin

             for ( int io=0; io<no; io++ )
               for ( int ii=0; ii<ni; ii++ )
                 wht[io][ii] = io < tinfo.no && ii < tinfo.ni ? $random : 0;

             for ( int ii=0; ii<ni; ii++ )
               ai[ii] = ii < tinfo.ni ? $random : 0;

             #1;

             for ( int io=0; io<tinfo.no; io++ ) begin

                if ( ao[0][io] !== ao[mut][io] ) begin
                   n_err++;
                   if ( n_err < 4 )
                     $write
                       ("Error test # %0d, output %0d: %0d != %0d (correct)\n",
                        tnum, io, ao[mut][io], ao[0][io] );
                end
             end

          end

          begin
             automatic string sat_key =
               $sformatf("Sat %0d",sat);
             automatic string long_key =
               $sformatf("%s %s",mname[mut],sat_key);
             automatic string msg =
               $sformatf("%0d %s tests on %s: %0d errors found.",
                         ntests, tinfo.name, mname[mut], n_err);
             $write("Done with %s\n",msg);
             testbench.test_summary =
               { testbench.test_summary, $sformatf("Results of %s\n", msg) };
             testbench.t_errs += n_err;
             testbench.t_errs_cat[{"All ",sat_key}] += n_err;
             testbench.t_errs_cat[long_key] += n_err;
             testbench.t_errs_cat[mname[mut]] += n_err;
          end

       end

    end

    done = 1;

 end


endmodule
```

```
// cadence translate_on
```

LSU EE 4755 Homework 4 Solution Due: 28 October 2020

☣ *Paper copies will not be accepted. E-mail your solution to* `koppel@ece.lsu.edu`. *A single PDF file is preferred.*

This assignment refers to the solution to Homework 3. Pieces are shown below, the complete solution can be found at `https://www.ece.lsu.edu/koppel/v/2020/hw03-sol.v.html` *and in the directory where the original assignment was copied from.*

This solution was prepared 3 Nov 2020 at 16:23. A more detailed solution may be posted later.

Problem 1: Using the simple model compute the cost and delay of the `nnAdd` module from Homework 3 (shown below) for both `sat=0` and `sat=1`. Do so after applying optimizations for constants. Show the cost and delay in terms of w. *Hint: See the simple model notes,* `https://www.ece.lsu.edu/koppel/v/2020/lsli-simple-model.pdf`*, for the cost of a ripple adder.*

- Show cost and delay in terms of w.
- Don't forget to optimize for constant values.
- Assume that the adder will be implemented using a ripple circuit.
- Indicate both the delay of the least-significant bit of the sum and the delay of the most significant bit of the sum. *Answering this part correctly and applying it to the other problems in this assignment will reveal something important about the impact of detecting overflow and of the different methods of doing so.*

```
module nnAdd #( int w = 5, sat = 0 )
              ( output uwire [w-1:0] so, input uwire [w-1:0] a, b );
   uwire [w:0] s = a + b;
   localparam logic [w-1:0] smax = ~w'(0);
   assign so = sat && s[w] ? smax : s[w-1:0];
endmodule
```

Under the simple model the cost of a w-bit ripple adder is $9w\,\mathrm{u_c}$, the delay of the least-significant bit is $4\,\mathrm{u_t}$ and the delay of the entire sum is $2(w+1)\,\mathrm{u_t}$.

For `sat=0` the cost and delay are those of the w-bit adder described above: The cost of the `sat=0` module is $9w\,\mathrm{u_c}$, the delay of the LSB in the `sat=0` module is $4\,\mathrm{u_t}$, and the delay of the MSB in the `sat=0` module is $2(w+1)\,\mathrm{u_t}$.

When `sat=1` the overflow logic must be taken into account too. That overflow logic synthesizes into a multiplexor with the select signal connected to `s[w]`, the zero input connected to `smax`, and the one input connected to the sum, `s[w-1:0]`. Because `smax` is a constant, the cost of the multiplexor is w and the added delay is 1. So, the cost with `sat=1` is $10w\,\mathrm{u_c}$.

Because the multiplexor control signal is connected to the carry out of the adder (which would be bit position w of the sum), *all* bits of the sum must wait for the MSB to arrive. That means that the delay for all bits in the `sat=1` module is $[2(w+1)+1]\,\mathrm{u_t}$. Sure, if all you cared about was the MSB this would be no big deal. But it precludes getting a faster result with cascaded ripple adders.

There are more problems on the next pages.

Problem 2: Using the simple model compute the cost and delay of the `nnMult` module from Homework 3 for `sat=1`. Let w denote the setting of both `wa` and `wb` (they are to be set to the same value), and let y denote the setting of `wp`. Solve this for $y < 2w$. Do so after applying optimizations for constants.

Solve this using the following cost for an unsigned integer multiplier with two w-bit inputs and a $2w$-bit output: the cost using the simple model is $10w^2\,\mathrm{u_c}$ and the delay is $[8w+2]\,\mathrm{u_t}$ for the complete product and $[4i+2]\,\mathrm{u_t}$ for bit position i. (The LSB is at position $i = 0$.) (For more details on how those were derived see the comments after the Linear Multiplier in `https://www.ece.lsu.edu/koppel/v/2020/mult-seq.v.html`.)

- Show the cost and delay in terms of w and y.
- Solve this for $y < 2w$.
- Don't forget to optimize for constant values.

```
module nnMult #( int wa = 5, wb = 6, wp = wa + wb, sat = 0 )
   ( output uwire [wp-1:0] p,  input uwire [wa-1:0] a, input uwire [wb-1:0] b );

   localparam logic [wp-1:0] pmax = ~wp'(0);
   localparam int wmx = wp > wa+wb ? wp : wa+wb;
   uwire [wmx-wp:0] phi;
   uwire [wp-1:0] plo;
   assign {phi,plo} = a * b;
   assign p = sat && wp < wa + wb && phi ? pmax : plo;

endmodule
```

In order to detect overflow the multiplier must compute a $2w$-bit product. The problem statement helpfully gives the cost of such hardware as $10w^2\,\mathrm{u_c}$ and the delay as $[8w+2]\,\mathrm{u_t}$.

The only difference with the saturation logic is that it must examine $2w-y$ bits of the product. If any of those $2w-y$ bits are 1 then there is overflow. As in the previous problem there is a multiplexor with the result (product in this case) at the zero input and `pmax` at the one input. The select signal is generated by ORing the $2w-y$ high bits of the product together. The cost of the multiplexor is $y\,\mathrm{u_c}$, and the cost of the OR gate is $[2w-y-1]\,\mathrm{u_c}$. Ordinarily under the simple model the delay for an a-input OR gate would be $\lceil \lg a \rceil$. But in this case we know the less significant bits of the product arrive earlier than the more significant bits. To implement an a-input OR gate for such a situation the OR gates can be connected linearly (rather than using a reduction tree). The MSB of the product would connect to the last OR gate, and so the delay for checking whether any of the $2w-y$ bits is 1 would just be $1\,\mathrm{u_t}$.

The total $\boxed{\text{cost of } \texttt{nnMult} \text{ is } [10w^2+y]\,\mathrm{u_c}}$ and $\boxed{\text{the delay of all bits is } [8w+2+1]\,\mathrm{u_t}}$.

There are more problems on the next pages.

Problem 3: Using the simple model determine the cost and performance of module `nn1xI` (shown on the next page) for the configurations described below. In all cases, let n denote the value of `ni`, w denote the value of `ww` and `wi` (which are the same) and y denote the value of `wo`. Assume the same hardware costs as the first two problems (modifying sizes and accounting for cascading where appropriate).

(a) Find the cost (not delay in this part) for `sat=0`, `tr=0`, and $y > 2w$ (that's one configuration) and for `sat=0`, `tr=1`, and $y > 2w$ (that's a second configuration). The two costs will be very similar.

- Show the costs in terms of n, w, and y.

Short answer: The cost for `tr=0` and `tr=1` is $[10nw^2 + 9(n-1)y]\,\mathrm{u_c}$ (or lower, see detailed answer).

Detailed Answer: The $y > 2w$ condition means that the multiplier will not overflow and that all bits of the product are needed. (If y were smaller, say $y = w$, then some of the adders used to implement the multiplier would be less than w bits and so would cost less.)

The `nnMADD` module consists of both an `nnAdd` and `nnMult` module. So the cost of the `tr=0` solution will be discussed in terms of the `nnAdd` and `nnMult` modules.

For both the `tr=0` and `tr=1` cases there will be n multipliers each having two w-bit inputs. The total cost of these multipliers is $10nw^2\,\mathrm{u_c}$.

For the `tr=0` case there are n `nnAdd` units but the input to the first adder is zero (because `s[-1]=0`), so after optimization there are $n-1$ adders. When n is a power of 2, the number of adders for the `tr=1` case is $\sum_{l=0}^{(\lg n)-1} 2^l = n-1$. So the number of adders is the same in both cases.

The code instantiates y-bit adders, but good synthesis programs—and good students—will have noticed that not all adders need y bits to avoid overflow. For the `tr=0` case the `i=1` adder has two $2w$-bit inputs and so only needs to compute a sum of $2w+1$ bits to avoid overflow. So if $y > 2w+1$ the cost can be reduced by using a $2w+1$ bit adder rather than a y bit adder. Call this a *trim optimization*.

First, compute the cost without the trim optimization. The cost of each of these adders is $9y\,\mathrm{u_c}$. The total cost of the adders for both `tr=0` and `tr=1` is $9y(n-1)\,\mathrm{u_c}$.

The total cost of the `nn1xI` module without the trim optimization is $[10nw^2 + 9(n-1)y]\,\mathrm{u_c}$.

The cost with the trim optimization will be computed for `tr=1`. Let l indicate a level in the recursion tree with l corresponding to a level in which $n = 2^l$. The base case is $l = 0$, for which there are no adders. For $l > 0$ there are $n/2^l$ adders each need $2w + l$ bits, so the cost at level l is $\left[\frac{n}{2^l}9(2w+l)\right]\,\mathrm{u_c}$. The total adder cost is $\sum_{l=1}^{(\lg n)} \left[\frac{n}{2^l}9(2w+l)\right]\,\mathrm{u_c} = [9(2w+2)(n-1) - 9\lg n]\,\mathrm{u_c}$.

(b) Find the delay (not cost in this part) for `sat=0`, `tr=0`, and $y > 2w$ (that's one configuration) and for `sat=0`, `tr=1`, and $y > 2w$ (that's a second configuration). The two delays will be very different.

- Show the delays in terms of n, w, and y.
- When computing the total delay don't forget to take into account the time that inputs arrive at each port, especially for the multiplier.
- When computing total delay account for cascading of ripple units.

At launch time ($t = 0$) inputs are available at all of the multipliers. As stated in the problem, bit i is correct at time $[4i+2]\,\mathrm{u_t}$.

First consider `tr=0`. For `i=1` (the `i` from the generate loop) the two inputs to the adder are from multipliers (because for `i=0` there is no need for an adder), and so bit i arrives at $4i+2$. Because the inputs to the adder aren't all available at the same time we can't rely on the ripple adder formula for when bit i of the sum is available. We know that each BFA requires 4 units of time to compute both the sum and carry output from its inputs when those inputs are available at the same time. Therefore, bit i of that first adder is available at $4i + 2 + 4 = 4i + 6$. Accounting for

$n-1$ adders, bit i is available at $4i+2+4(n-1) = 4(i+n)-2$ and the most-significant bit, $y-1$ is available at $[4(y-1)+4n-2]\,\mathrm{u_t} = [4(y+n)-6]\,\mathrm{u_t}$.

So, the delay for the LSB when `tr=0` is $[4n-2]\,\mathrm{u_t}$ and the delay for the MSB when `tr=0` is $[4(y+n)-6]\,\mathrm{u_t}$.

For `tr=1` the computation is similar, except that the critical path passes through $\lg n$ adders rather than $n-1$ adders. Therefore the delay for bit i is $[4i+2+4\lg n]\,\mathrm{u_t}$ and the MSB is available at $[4(y-1)+2+4\lg n]\,\mathrm{u_t}$.

So, the delay for the LSB when `tr=1` is $[2+4\lg n]\,\mathrm{u_t}$ and the

delay for the MSB when `tr=1` is $[4(y-1)+2+4\lg n]\,\mathrm{u_t}$.

(*c*) Find the delay for `sat=1`, `tr=0`, and $y > 2w$ (that's one configuration) and for `sat=1`, `tr=1`, and $y > 2w$ (that's a second configuration). The two delays should be very different from each other and from the delays from the previous problem.

Since $y > 2w$ there will be no saturation penalty for the multiplier. Therefore bit i of the product is stable at $4i+2$.

Because of the multiplexor, the delay though one saturating adder (an **`nnAdd`** module with `sat=1`) is the same for all bits. That delay is $[2(y+1)+1]\,\mathrm{u_t}$.

First consider `tr=0`. For `i=1` (the genvar, not a bit position) bit i of the sum (before saturation, and accounting for the multiplier delay) is ready at time $4i+2+4 = 4i+6$. The MSB, $y-1$, is available at $4y+2$, and the output of the mux is available at $[4y+3]\,\mathrm{u_t}$. The delay computation is different for the remaining $n-2$ adders. Consider the `i=2` (the genvar) adder. One input is from the `i=1` adder and the other is from a multiplier. The input from the `i=1` adder arrives at $[4y+3]\,\mathrm{u_t}$ (which we just calculated). By then all bits from the multiplier will have arrived. So the time at which the LSB can be computed is $4y+3$, there is no early start. The sum will be computed $2(y+1)$ later, or at a total delay of $[4y+3+2(y+1)+1]\,\mathrm{u_t} = [6y+5+1]\,\mathrm{u_t}$ including the mux. The complete sum is available at $[4y+3+(n-2)(2(y+1)+1)]\,\mathrm{u_t}$ or $[n(2y+3)-3]\,\mathrm{u_t}$.

Notice that with without saturation the time is $O(n+y)$ and that with saturation the time is $O(ny)$, much worse!

The computation is similar for the `tr=1` case. The critical path starts with a multiply, add, saturate (same as for `tr=0`) with a delay of $[4y+3]\,\mathrm{u_t}$. After that the critical path passed through $(\lg n)-1$ additional adders, so the total time is $[4y+3+(\lg n-1)(2(y+1)+1)]\,\mathrm{u_t}$ or $[(2y+3)\lg n+2y]\,\mathrm{u_t}$. Here the time is order $O(y\lg n)$ which is better than $O(ny)$ but not nearly as good as $O(n+y)$.

```
module nn1xI #( int wo = 10, wi = 4, ww = 5, ni = 2, tr = 0, sat = 0 )
   ( output uwire [wo-1:0] ao,
     input uwire [wi-1:0] ai[ni],
     input uwire [ww-1:0] wht[ni] );

   if ( tr ) begin

      if ( ni == 1 ) begin

         nnMult #(wi,ww,wo,sat) mult(ao, ai[0], wht[0] );

      end else begin

         localparam int nlo = ni / 2;
         localparam int nhi = ni - nlo;
         uwire [wo-1:0] aolo, aohi;
         nn1xI #(wo,wi,ww,nlo,1,sat) nnlo(aolo, ai[0:nlo-1], wht[0:nlo-1]);
         nn1xI #(wo,wi,ww,nhi,1,sat) nnhi(aohi, ai[nlo:ni-1], wht[nlo:ni-1]);
         nnAdd #(wo,sat) add(ao,aolo,aohi);

      end

   end else begin

      uwire [wo-1:0] s[ni-1:-1];
      assign s[-1] = 0;
      assign ao = s[ni-1];

      for ( genvar i = 0;  i < ni;  i++ )
        nnMADD #(ww,wi,wo,sat) madd( s[i], wht[i], ai[i], s[i-1] );

   end

endmodule

module nnMADD #( int wa = 10, wb = 5, ws = wa + wb, sat = 0 )
   ( output uwire [ws-1:0] so,
     input uwire [wa-1:0] a, input uwire [wb-1:0] b, input uwire [ws-1:0] si);

   uwire [ws-1:0] p;
   nnMult #(wa,wb,ws,sat) mu(p, a, b);
   nnAdd #(ws,sat) ad(so, si, p);

endmodule
```

There are even more problems on the next pages.

Problem 4: Consider module `nn0xI` instantiated with `no=1`, `tr=0`, for both `sat=1` and `sat=2`. (A slightly simplified version appears below.) Let n denote the value of `ni`, w denote the value of `wi` and `ww` (which are the same), and let y denote the value of `wo`.

Assume that $2w < y < \lceil \lg n(2^w - 1)^2 \rceil$. That is, y is large enough so that the multipliers can't overflow but not so large that the adders can't overflow.

(a) Compute the cost and delay for both the `sat=1` and `sat=2` cases. For `sat=1` just re-use answers from the previous problems.

- Show answers in terms of n, w, and y.
- Don't forget that the value of `wo` in the `nn1xI` instantiations depends upon `sat`.

When `sat=1` and `no=1` the hardware for `nn0xI` is the same as that of `nn1xI`.

The cost of the `sat=0` instantiation based on the answer to Problem 3 is $[10nw^2 + 9(n-1)y]\,\mathrm{u_c}$. The cost of the saturation hardware is that of $n-1$ 2-input, y-bit multiplexors in which one input is a constant. The cost of these is $[(n-1)y]\,\mathrm{u_c}$. So the total cost for `nn0xI` with `sat=1` is $[10nw^2 + 9(n-1)y + (n-1)y]\,\mathrm{u_c} = [10nw^2 + 10(n-1)y]\,\mathrm{u_c}$.

The delay has been computed in Problem 3, it is $[n(2y+3) - 3]\,\mathrm{u_t}$.

When `sat=2` the `nn1xI` modules are instantiated with `sat=0`, and so their cost and delay are $[10nw^2 + 9(n-1)r]\,\mathrm{u_c}$ and $[4(r+n) - 6]\,\mathrm{u_t}$ where r is the value of `wr` for which they were instantiated. Note that $r = \lceil \lg n(2^w - 1)^2 \rceil$.

Module `nn0xI` checks for saturation by checking whether the high $r - y$ bits of `ar` are non-zero. That can be done using an OR gate, with a cost of $[r - y - 1]\,\mathrm{u_c}$. If the OR used a tree reduction the delay would be $\lg(r-y)\,\mathrm{u_t}$, but if we expect bit $i+1$ to arrive at least one $\mathrm{u_t}$ later than bit i a linear connection of OR gates would be faster, and have a net delay of just 1. So the total delay is $[4(r+n) - 6 + 1]\,\mathrm{u_t}$.

The total cost includes a 2-input, y-bit multiplexor and the $(r-y)$-input OR gate. One input to the mux is constant, so its cost is y. The total cost with this hardware is $[10nw^2 + 9(n-1)r + y + (r - y - 1)]\,\mathrm{u_c}$ or $[10nw^2 + 9(n-1)r + r - 1]\,\mathrm{u_c}$ where $r = \lceil \lg n(2^w - 1)^2 \rceil$.

(b) In terms of the costs computed above is `sat=2` always better, always worse, or sometimes better than `sat=1`? Be specific of course.

Recall that for `sat=1` the cost is $C(1,n,w,y) = [10nw^2 + 10(n-1)y]\,\mathrm{u_c}$ and for `sat=2` the is $c(2,n,w,y) = [10nw^2 + 9(n-1)r + r - 1]\,\mathrm{u_c}$.

To solve this compute $C(1,n,w,y) - C(2,n,w,y)$. If the result is always positive then `sat=2` always costs less, etc.

$C(1,n,w,y) - C(2,n,w,y) = 10(n-1)y - 9(n-1)r - r + 1$

Recall $r = \lceil \lg n(2^w - 1)^2 \rceil$ and that the assumption is that $2w < y < r$. We can approximate $r \approx 2w + \lg n$.

The cost benefit for `sat=2` is less favorable larger when y is smaller. Consider one minus the smallest value of y, which is $y = 2w$ Then $C(1,n,w,y) - C(2,n,w,y) = 10(n-1)2w - 9(n-1)(2w + \lg n) - (2w + \lg n) + 1 = (n-1)2w - 9(n-1)\lg n - 2w - \lg n + 1 \approx (n-1)2w - 9(n-1)\lg n$. This expression is positive when $w > 2.25 \lg n$. Generally when w is large `sat=2` works better, when n is large `sat=1` works better.

```
module nn0xI #( int no = 4, ni = 2, wo = 10, wi = 4, ww = 5, tr = 0, sat = 0 )
   ( output uwire [wo-1:0] ao[no],
     input uwire [wi-1:0] ai[ni],   input uwire [ww-1:0] wht[no][ni] );

   // Compute number of bits to represent largest possible value that
   // can appear on an ao.
   localparam int wr = $clog2( ( 2**wi - 1 ) * ( 2**ww  - 1 ) * ni );

   if ( sat < 2 ) begin
```

```
    for ( genvar i = 0;  i < no;  i++ )
      nn1xI #(wo,wi,ww,ni,tr,sat) row( ao[i], ai, wht[i] );

  end else begin

    for ( genvar i = 0;  i < no;  i++ ) begin

      uwire [wr-1:0] ar;
      nn1xI #(wr,wi,ww,ni,tr,0) row( ar, ai, wht[i] );
      assign ao[i] = ar[wr-1:wo] ? ~wo'(0) : ar[wo-1:0];

    end

  end

endmodule
```

Problem 5: *Zero points will be given for the answer to this question, but please try your very best to answer it.* Suggest a method of saturating `ao` that avoids the extra `wo` bits needed (for `nn1xI`) when `sat=2` but also avoids the critical-path-killing saturation logic used when `sat=1`. Your solution could add extra ports to all modules except `nn0xI`. A correct solution would detect overflow under the same conditions as `nn0xI` does with `sat=1`.

19 Fall 2019 Solutions

LSU EE 4755 Homework 1 Solution Due: 18 September 2019

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/2019/hw01.v.html`.

Problem 0: Following instructions at `https://www.ece.lsu.edu/koppel/v/proc.html`, set up your class account, copy the assignment, and run the Verilog simulator and synthesis program on the unmodified homework file, `hw01.v`. Do this early enough so that minor problems (*e.g.*, password doesn't work) are minor problems.

Homework Overview

In class you were told that for common operations, such as shifting, addition, and multiplication, it's better to use Verilog operators in procedural code than to re-invent the wheel by writing Verilog to implement those operations. This point was made when covering the shift module in the introductory lectures. For example, if you need a shifter it's better to just use the shift operator:

```
module shift_right_operator
  ( output uwire [15:0] shifted,
    input uwire [15:0] unshifted, input uwire [3:0] amt );
   assign shifted = unshifted >> amt;
endmodule
```

than to write code for your own shifter:

```
module shift_right_logarithmic
  ( output uwire [15:0] sh, input uwire [15:0] s0, input uwire [3:0] amt );
   uwire [15:0] s1, s2, s3;
   mux2 st0( s1, amt[0], s0, {1'b0, s0[15:1]} );
   mux2 st1( s2, amt[1], s1, {2'b0, s1[15:2]} );
   mux2 st2( s3, amt[2], s2, {4'b0, s2[15:4]} );
   mux2 st3( sh, amt[3], s3, {8'b0, s3[15:8]} );
endmodule

module mux2( output uwire [15:0] x,
             input uwire select, input uwire [15:0] a0, a1 );
    assign x = select ? a1 : a0;
endmodule
```

The reason for showing the implementation of shifters, and other common operations, was to teach general design concepts using operations that you should be familiar with. That will be the approach in this homework, in which a multiplier is to be implemented.

Testbench Code

The testbench for this assignment, which can be run when visiting the file in Emacs in a properly set-up account by pressing F9, tests the multiply modules. Modules `mult_operator` and `mult16` should pass, `mult16_tree` awaits your solution. A sample of the end of the testbench output appears below:

```
Starting testbench...
Error in mult16_tree test    0:  xxxxxxxx != 00000001 (correct)
Error in mult16_tree test    1:  xxxxxxxx != 00000002 (correct)
Error in mult16_tree test    2:  xxxxxxxx != 00000020 (correct)
Error in mult16_tree test    3:  xxxxxxxx != 00000020 (correct)
```

```
Error in mult16_tree test    4:  xxxxxxxx != 139dff24 (correct)
Error in mult16_tree test    5:  xxxxxxxx != 4839cb7b (correct)
Mut mult_operator  ,    0 errors (0.0% of tests)
Mut mult16_flat    ,    0 errors (0.0% of tests)
Mut mult16_tree    , 1000 errors (100.0% of tests)
Memory Usage - 38.6M program + 154.6M data = 193.2M total
CPU Usage - 0.0s system + 0.0s user = 0.1s total (70.4% cpu)
Simulation complete via $finish(2) at time 10 US + 0
./hw01.v:218       $finish(2);
ncsim> exit
```

A count of the number of tests and errors is shown for three modules. The testbench shows the first six errors it finds on each module. To see more than six modify the testbench (search for `err_limit`). In the output above the testbench is showing that the module outputs are `x` (uninitialized) which of course don't match the expected outputs.

Use Simvision to debug your modules. Feel free to modify the testbench so that it presents inputs that facilitate debugging.

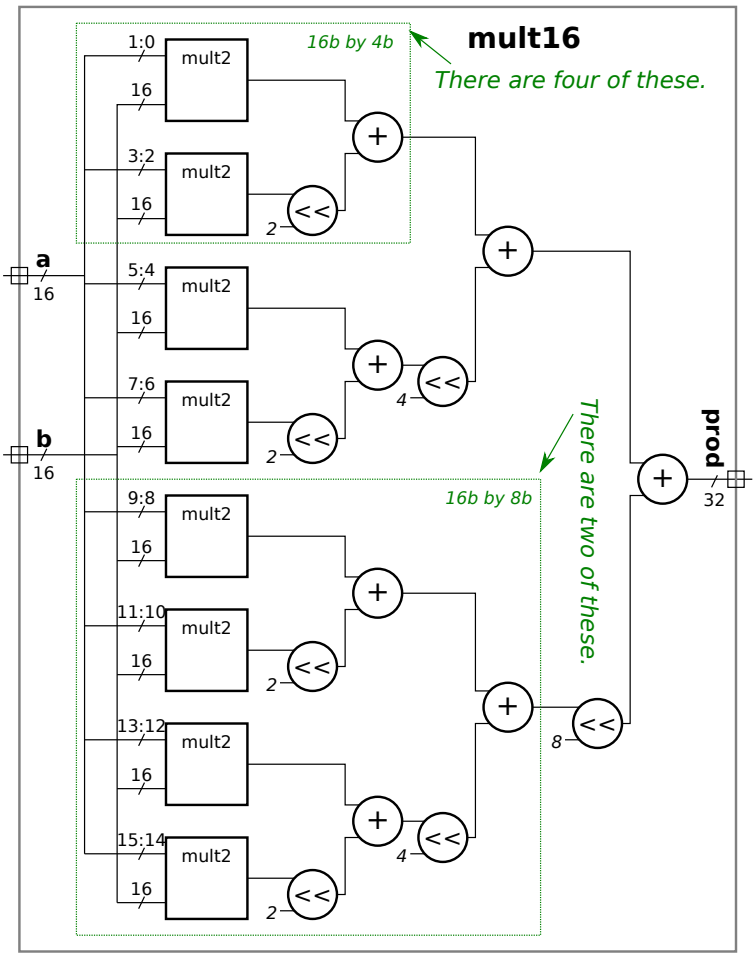
Synthesis

The synthesis script, `syn.tcl`, will synthesize the three modules each with two delay targets, an easy 10 ns and a un-achievable 0.1 ns. If the module doesn't synthesize −.001 s is shown for the delay. The script is run using the shell command `genus -files syn.tcl`, which invokes Cadence Genus.

The synthesis script shows area (cost), delay, and the delay target in a neat table. Additional output of the synthesis program is written to file `spew.log`. Sample synthesis script output appears below:

Problem 1 on next page.

Problem 1: The illustration to the right shows a sketch of a multiplier, `mult16`, with two 16-bit inputs and a 32-bit output. The multiplier is constructed from `mult2` modules, shifters (<<), and adders. The illustrated module is similar to the multiplier in `mult16_flat` in `hw01.v`. The `mult2` modules have two inputs, one is two bits, the other is 16 bits. Each input holds an unsigned integer. The output, 18 bits, is the product of the two inputs. Notice that each `mult2` module is connected to two bits of `a` and all bits of `b`. The outputs of the `mult2` modules are shifted and added together in such a way that `prod` is the correct product of `a` and `b`.



There are two parts of `mult16` surrounded by green boxes. The upper one, labeled *16b by 4b*, contains two `mult2` modules. The label is explaining that the boxed material multiplies a 16-bit number by a 4-bit number. A similar box could have been put around the next pair of `mult2` modules, etc. The hardware within each of these four boxes would be identical. (The bit slices at the upper `mult2` inputs, such as `1:0` and `5:4` are different, but that can be taken care of outside the green box.) Think about the poor soul who might have just typed in all the Verilog for `mult16` and then suddenly realizes this. All that person would have had to do would be to code one module, call it `mult4_tree`, and just instantiate it four times. Here is an almost empty version of `mult4_tree`:

```
module mult4_tree
  ( output uwire [0:0] prod, // Need to change output size.
    input uwire [3:0] a, input uwire [15:0] b );

   mult2 mlo( /* finish */ );
   mult2 mhi( /* finish */ );

endmodule
```

Alert students might suspect that we don't actually instantiate `mult4_tree` *four* times because the *16b by 8b* section itself could be a module which would contain only two instantiations of `mult4_tree`. That would be correct.

Modify modules `mult16_tree`, `mult8_tree`, and `mult4_tree` found in `hw01.v` so that they implement the multiplier described above. Module `mult16_tree` must instantiate exactly two `mult8_tree` modules, module `mult8_tree` must instantiate exactly two `mult4_tree` modules, and

`mult4_tree` must use the two `mult2` modules that are already instantiated (but with the ports missing).

In each module use implicit structural code or behavioral code to combine the outputs of that module's two instantiated modules. It might be helpful to look at `mult16_flat` for examples of instantiation and implicit procedural code.

Start with module `mult16_tree`. You can test your changes to `mult16_tree` by putting placeholder code in `mult8_tree`, such as `assign prod = a*b;`. Don't forget to change the port sizes on `mult8_tree` to what they should be based on the diagram.

Once the testbench reports zero errors move the placeholder to `mult4_tree` and complete `mult8_tree`. Continue until the three modules are finished.

Some of the port sizes are set to 1 bit, `[0:0]`. Those are placeholders, change those to the correct sizes, but no larger. Credit will be deducted for oversized ports, especially if all ports are made 32 bits.

Pay attention to port-size warnings when running the simulator.

The solution Verilog code has been placed in the assignment directory, and on the Web at `https://www.ece.lsu.edu/koppel/v/2019/hw01-sol.v.html`.

To solve the problem one needed to see that `a` was split between the two modules, `mlo` and `mhi`, but that a complete version of `b` was used in each. Another important element to work out was the size of the product. When an x-bit unsigned integer is multiplied by a y-bit unsigned integer, the maximum sized product is $x + y$ bits. So the `mult8_tree` output, and the wire that connects to it, must be $8 + 16 = 24$ bits. Therefore in the solution (shown below) `prod_lo` and `prod_hi` are 24 bits, as is the output of the `mult8_tree` module.

```
module mult16_tree
  #( int wa = 16, int wb = 16, int wp = wa + wb )
   ( output uwire [31:0] prod, input uwire [15:0] a, input uwire [15:0] b );

   /// SOLUTION

   // Declare properly-sized connections to mult8_tree outputs.
   uwire [23:0] prod_lo, prod_hi;

   // Instantiate two mult8_tree multipliers, each handles 8 bits of a.
   mult8_tree mlo( prod_lo, a[7:0],  b);
   mult8_tree mhi( prod_hi, a[15:8], b);

   // Compute the full product using the two partial products.
   assign prod = prod_lo + ( prod_hi << 8 );

endmodule

module mult8_tree
  ( output uwire [23:0] prod,
    input uwire [7:0] a, input uwire [15:0] b );
   /// SOLUTION
   uwire [19:0] prod_lo, prod_hi;
   mult4_tree mlo( prod_lo, a[3:0], b);
   mult4_tree mhi( prod_hi, a[7:4], b);
   assign prod = prod_lo + ( prod_hi << 4 );
endmodule

module mult4_tree
```

```
  ( output uwire [19:0] prod,
    input uwire [3:0] a, input uwire [15:0] b );
   /// SOLUTION
   uwire [17:0] prod_lo, prod_hi;
   mult2 mlo( prod_lo, a[1:0], b);
   mult2 mhi( prod_hi, a[3:2], b);
   assign prod = prod_lo + ( prod_hi << 2 );
endmodule
```

Problem 2: The synthesis script will synthesize `mult16_tree` from Problem 1, plus two already working modules, `mult16_flat` and `mult_operator`, which just uses the multiply operator.

If the synthesis program were perfect then all three modules would have the same cost and delay because they each do exactly the same thing (multiply) and so the optimization algorithms would have found the same lowest-cost circuit from each one. Spoiler alert: Genus is not perfect.

Guess which module you think will be the fastest or least expensive, and explain why. Then run the synthesis script and comment on whether the results met your expectations.

Solution on next page.

I would expect that **mult_operator** would be fastest with the 0.1 ns delay target and least expensive with the 10 ns target because integer multiplication is a common operation and so the synthesis program should have a well-tuned multiply module in its library for situations such as these.

If optimization was not very good, then I'd expect **mult16_flat** to have a longer delay than **mult16_tree** because of the expression adding together the partial products:

```
assign prod = prod00 + ( prod02 << 2 ) + ( prod04 << 4 ) + ( prod06 << 6 ) + ( prod08 <<
8 ) + ( prod10 << 10 ) + ( prod12 << 12 ) + ( prod14 << 14 );
```

This expression has seven additions. If the order of additions follows the expression above then each addition after the first will not have its operands ready until the previous addition finishes. Therefore the critical path passes through seven additions. In the tree version the critical pass passes through just three additions, and so would be faster.

Modern optimizers, however, should be able to *re-associate* the expression to reduce the critical path. For example, internally the optimizer might convert the expression into:

```
assign prod =
       (
         ( ( prod00 )       + ( prod02 << 2 ) )
         +
         ( ( prod04 << 4  ) + ( prod06 << 6 ) )
       )
       +
       (
         ( ( prod08 << 8  ) + ( prod10 << 10 ) )
         +
         ( ( prod12 << 12 ) + ( prod14 << 14 ) )
       );
```

In the expression above the four inner additions (the ones where the plus sign is in the middle of the line) can start at the same time, when they finish two more additions can start and proceed in parallel, followed by the last addition in the center of the expression.

Below is the actual synthesis output:

Module Name	Area	Delay Actual	Delay Target
mult_operator	235272	9.266	10.000 ns
mult16_flat	403519	9.982	10.000 ns
mult16_tree	294419	8.861	10.000 ns
mult_tree	240616	7.934	10.000 ns
mult_operator_1	491053	3.103	0.100 ns
mult16_flat_1	817229	4.502	0.100 ns
mult16_tree_1	590500	3.360	0.100 ns
mult_tree_3	510150	3.150	0.100 ns

The results indicate that optimizers are not as good as I thought. As expected, the library routine, and so **mult_operator** was least expensive. But **mult_tree** was almost as good, and for some reason was better than **mult16_tree**, perhaps because it does not use a multiplier in its terminal case. For delay the library routine also wins out and our tree-structured modules outperform the flat ones.

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2019 Homework 1 -- SOLUTION
//

 /// Assignment https://www.ece.lsu.edu/koppel/v/2019/hw01.pdf


`default_nettype none

////////////////////////////////////////////////////////////////////////////////
/// Problem 1 -- SOLUTION
//
 /// Modify mult16_tree, mult8_tree, and mult4_tree to implement multiplier.
 ///
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] mult16_tree must use exactly two mult8_tree modules, etc.
//     [✔] Pay attention to port widths. Do not make them larger than needed.
//     [✔] Module must be synthesizable. Use command: genus -files syn.tcl


module mult16_tree
  #( int wa = 16, int wb = 16, int wp = wa + wb )
   ( output uwire [31:0] prod,
     input uwire [15:0] a,
     input uwire [15:0] b );

   /// Problem 1 solution goes here, and in other modules.
   //  [✔] Instantiate two mult8_tree's.
   //  [✔] Use implicit structural or behavioral code to combine their outputs.

   /// SOLUTION

   // Declare properly-sized connections to mult8_tree outputs.
   //
   uwire [23:0] prod_lo, prod_hi;
   //
   // They are 24 bits wide because that's the maximum size of the
   // product of an 8-bit unsigned integer (such as a[7:0]) and a
   // 16-bit unsigned integer (b): 8+16 =24.

   // Instantiate two mult8_tree multipliers, each handles 8 bits of a.
   //
   mult8_tree mlo( prod_lo, a[7:0],  b);
   mult8_tree mhi( prod_hi, a[15:8], b);

   // Compute the full product using the two partial products.
   //
   assign prod = prod_lo + ( prod_hi << 8 );
   //
   // Because prod is 32-bits wide the right-hand side computation
   // will be computed with a 32-bit precision.

endmodule

module mult8_tree
  ( output uwire [23:0] prod,
    input uwire [7:0] a,
    input uwire [15:0] b );
   //  [✔] Pay attention to port widths. Do not make them larger than needed.

   /// Problem 1 solution goes here, and in other modules.
   //  [✔] Instantiate two mult4_tree's.
   //  [✔] Use implicit structural or behavioral code to combine their outputs.

   /// SOLUTION
   //
   // See the solution comments description in mult16_tree.

   uwire [19:0] prod_lo, prod_hi;
   mult4_tree mlo( prod_lo, a[3:0], b);
   mult4_tree mhi( prod_hi, a[7:4], b);
   assign prod = prod_lo + ( prod_hi << 4 );

endmodule

module mult4_tree
  ( output uwire [19:0] prod,
    input uwire [3:0] a,
```

```
    input uwire [15:0] b );
   //  [✔] Pay attention to port widths. Do not make them larger than needed.

   /// Problem 1 solution goes here, and in other modules.
   //  [✔] Use implicit structural or behavioral code to combine their outputs.

   /// SOLUTION
   //
   // See the solution comments description in mult16_tree.

   uwire [17:0] prod_lo, prod_hi;

   mult2 mlo( prod_lo, a[1:0], b);
   mult2 mhi( prod_hi, a[3:2], b);
   assign prod = prod_lo + ( prod_hi << 2 );

endmodule


 /// Bonus Solution:
module mult_tree
  #( int wa = 16, int wb = 16, int wp = wa + wb )
   ( output uwire [wp:1] prod,
     input uwire [wa:1] a,
     input uwire [wb:1] b );

   /// BONUS SOLUTION
   //
   //  This answers a question that was almost but not quite asked:
   //    Using generate statements design a single module that can be
   //    instantiated into a module equivalent to mult16_tree,
   //    mult8_tree, mult4_tree, and mult2, and also mult32_tree, etc.

   if ( wa == 1 ) begin

      // Terminal case: 1 bit partial product.
      //
      assign prod = a ? b : 0;
      //
      // Equivalent to: prod = a * b;

   end else begin

      // Split a in half and recursively instantiate a module for each
      // half.

      localparam int wn = wa / 2;
      localparam int wx = wb + wn;

      uwire [wx:1] prod_lo, prod_hi;

      mult_tree #(wn,wb) mlo( prod_lo, a[wn:1], b);
      mult_tree #(wn,wb) mhi( prod_hi, a[wa:wn+1], b);

      // Combine the partial products.
      //
      assign prod = prod_lo + ( prod_hi << wn );

   end

endmodule



 /// Do not modify the code below this point.

module mult2
  ( output uwire [17:0] prod, input uwire [1:0] a, input uwire [15:0] b );

   /// DO NOT MODIFY THIS ROUTINE.
   assign prod = a * b;

endmodule


module mult16_flat
  #( int wa = 16, int wb = 16, int wp = wa + wb )
   ( output uwire [31:0] prod, input uwire [15:0] a, b );

   /// DO NOT MODIFY THIS ROUTINE.
```

```
`ifdef NEVER_DEFINE_ME
   // Emacs Lisp code to generate Verilog code for mult16_flat.
   (cl-loop for i from 0 to 14 by 2
    concat (if (= i 0) "   assign prod = prod00"
             (format " + ( prod%02d << %d )" i i)) into prod
    concat (format "%s prod%02d" (if (= i 0) "" ",") i) into decl
    concat (format "   mult2 m%d( prod%02d, a[%d:%d], b);\n" i i (+ i 1) i)
    into inst
    finally (insert (concat "\n   uwire [17:0]" decl ";\n" inst "\n" prod ";\n")))
`endif

   uwire [17:0]  prod00, prod02, prod04, prod06, prod08, prod10, prod12, prod14;
   mult2 m0( prod00, a[1:0], b);
   mult2 m2( prod02, a[3:2], b);
   mult2 m4( prod04, a[5:4], b);
   mult2 m6( prod06, a[7:6], b);
   mult2 m8( prod08, a[9:8], b);
   mult2 m10( prod10, a[11:10], b);
   mult2 m12( prod12, a[13:12], b);
   mult2 m14( prod14, a[15:14], b);

   assign prod = prod00 + ( prod02 << 2 ) + ( prod04 << 4 ) + ( prod06 << 6 ) + ( prod08 << 8 ) + ( prod10 << 10 ) + ( prod12 << 12 ) + ( 

endmodule



module mult_operator
  #( int wa = 16, int wb = 16, int wp = wa + wb )
   ( output uwire [wp:1] prod, input uwire [wa:1] a, input uwire [wb:1] b );
   /// DO NOT MODIFY THIS ROUTINE.
   assign prod = a * b;
endmodule




////////////////////////////////////////////////////////////////////////////////
/// Testbench Code

// cadence translate_off

module testbench;

   localparam int wid = 16;
   localparam int num_tests = 1000;
   localparam int NUM_MULT = 4;
   localparam int err_limit = 7;

   logic [wid-1:0] plier, cand;
   logic [2*wid-1:0] prod[NUM_MULT], shadow_prod;

   mult_operator mb0(prod[0], plier, cand);
   mult16_flat mb1(prod[1], plier, cand);
   mult16_tree mb2(prod[2], plier, cand);
   multw_tree #(wid,wid) mb3(prod[3], plier, cand);

   string names[] = '{ "mult_operator", "mult16", "tree16", "treep"  };

   int err_cnt[NUM_MULT];

   // Array of multiplier/multiplicand values to try out.
   // After these values are used a random number generator will be used.
   //
   int tests[$] = {1,1, 1,2,  1,32,  32, 1};

   initial begin

      $display("Starting testbench.\n");

      for ( int i=0; i<num_tests; i++ ) begin

         // Set multiplier and multiplicand values.
         //
         plier = tests.size() ? tests.pop_front() : $random();
         cand = tests.size() ? tests.pop_front() : $random();

         shadow_prod = plier * cand;

         #10;
```

```
         // Make sure each module's output is correct.
         //
         for ( int mut=0; mut<NUM_MULT; mut++ ) begin

            if ( shadow_prod !== prod[mut] ) begin

               err_cnt[mut]++;

               if ( err_cnt[mut] < err_limit )
                 $display("Error in %s test %4d:  %x != %x (correct)\n",
                          names[mut], i, prod[mut], shadow_prod);
            end

         end

      end

      // Tests completed, report error count for each device.
      //
      for ( int mut=0; mut<NUM_MULT; mut++ ) begin

         $display("Mut %s, %d errors (%.1f%% of tests)\n",
                  names[mut], err_cnt[mut],
                  100.0 * err_cnt[mut]/real'(num_tests) );

      end

      $finish(2);

   end

endmodule

// cadence translate_on
```

LSU EE 4755 Homework 2 Solution Due: 8 October 2019

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/2019/hw02.v.html`.

Problem 0: Following instructions at `https://www.ece.lsu.edu/koppel/v/proc.html`, set up your class account, copy the assignment, and run the Verilog simulator and synthesis program 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 Correction (December 2019)

When assigned in October 2019 this assignment defined clz backward, starting at the least-significant bit. That has been corrected in this version and in the posted code.

Homework Overview

A *count leading zeros (clz)* operation returns the number of consecutive zeros starting at the most significant bit of an integer's binary representation. For example, the clz of 00101_2 is 2, the clz of 101_2 is 0, and the clz of 32-bit number 0_2 is 32. The Verilog module below computes the clz of its input:

```
module clz
  #( int w = 19, int ww = $clog2(w+1) )
   ( output var logic [ww-1:0] nlz, input uwire logic [w-1:0] a );

   uwire [w:0] aa = { a, 1'b1 };
   always_comb for ( int i=0; i<=w; i++ ) if ( aa[i] ) nlz = w-i;
endmodule
```

The module was written as behavioral code, but it does turn out to be synthesizable. Nevertheless, one may wonder if the synthesis program will do a good job with this. (Later in the semester we will learn what kind of hardware will be inferred for the description above.) One way to find out is to design a module which *should* be efficient and see how well it compares to what the synthesis program does with the module above. That, and the use of generate statements, is the subject of this assignment.

Testbench Code

The testbench for this assignment, which can be run when visiting the file in Emacs in a properly set-up account by pressing F9, tests the `clz_tree` module at several different widths. All should initially fail. A shortened sample of the testbench output appears below:

```
ncsim> run
** Starting tests for width 1.
Error for width  1: input 1:  z != 0 (correct).
Error for width  1: input 0:  z != 1 (correct).
Error for width  1: input 1:  z != 0 (correct).
Error for width  1: input 0:  z != 1 (correct).
Width 1, done with 10 tests, 10 errors.
** Starting tests for width 2.
Error for width  2: input 3:  z != 0 (correct).
Width 2, done with 20 tests, 20 errors.
** Starting tests for width 5.
```

```
[snip]
Error for width 17: input 08959:   z != 0 (correct).
Width 17, done with 170 tests, 170 errors.
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
Total number of errors: 610
```

The testbench prints the details of the first four errors it finds, and after that prints just one detail time per width. A total for each width and a grand total are printed, see the transcript above.

Use Simvision to debug your modules. Feel free to modify the testbench so that it presents inputs that facilitate debugging.

Synthesis

The synthesis script, `syn.tcl`, will synthesize `clz` (for reference) and `clz_tree` (your solution). Each module will be synthesized at three widths, and with two delay targets, an easy 10 ns and a un-achievable 0.1 ns. If a module doesn't synthesize −.001 s is shown for its delay. The script is run using the shell command `genus -files syn.tcl`, which invokes Cadence Genus. If you would like to synthesize additional modules or sizes edit `syn.tcl` near the bottom.

The synthesis script shows area (cost), delay, and the delay target in a neat table. Additional output of the synthesis program is written to file `spew.log`.

Problem 1 on next page.

Problem 1: Complete module `clz_tree` so that it computes the clz of its input in a tree-like fashion. For the non-terminal case it should instantiate two `clz_tree` modules and each should operate on part of the input, `a`. The outputs of these two modules should be appropriately combined. To help you get started, a recursive solution to Homework 1, `mult_tree`, is in `hw02.v`.

An easy mistake to make is using the wrong sized variable in a module port connection. Previously the Verilog software (`ncelab` to be precise) would issue a warning which was easy to miss. Now a port size mismatch is a fatal error.

For maximum credit do not use adders in your design. Adders can be avoided if the size of the low module is always a power of 2.

See the Verilog code check boxes for additional items to check for.

The solution appears below. The partial-credit solution, using an adder, appears first.

```
/// SOLUTION – With Adder. Two points would be deducted.
module clz_tree_fat
  #( int w = 19, int ww = $clog2(w+1) )
   ( output uwire [ww:1] nlz, input uwire [w:1] a );

   if ( w == 1 ) begin

      assign nlz = ~ a;

   end else begin

      localparam int wlo = w/2;
      localparam int whi = w - wlo;
      localparam int wwlo = $clog2(wlo+1);
      localparam int wwhi = $clog2(whi+1);

      uwire [wwlo:1] lz_lo;
      uwire [wwhi:1] lz_hi;

      clz_tree_fat #(wlo) clo( lz_lo, a[wlo:1]   );
      clz_tree_fat #(whi) chi( lz_hi, a[w:wlo+1] );

      assign nlz = lz_hi < whi ? lz_hi : whi + lz_lo;

   end

endmodule
```

The better solution, without the adder, is on the next page.

The solution below avoids an adder by setting the size of the hi module to a power of 2. If all of the high bits are zero, then the clz is the count of the number of low bits, plus a power of 2. The power of 2 to add is parameter **lhi** (see the code).

/// SOLUTION – Without Adder

```
module clz_tree #( int w = 19, int ww = $clog2(w+1) )
                 ( output uwire [ww-1:0] nlz, input uwire [w-1:0] a );
   if ( w == 1 ) begin
      assign nlz = !a[0];
   end else if ( w == 2 ) begin
      assign nlz = { !a[0] && !a[1], !a[1] && a[0] };
   end else begin

      // Set whi to the largest power of 2 strictly less than w.
      //
      localparam int lhi = $clog2(w) - 1;
      localparam int whi = 1 << lhi;
      localparam int wwhi = lhi + 1;

      // Then set wlo to the number of remaining bits.
      //
      localparam int wlo = w - whi;

      uwire [wwhi-1:0] nlz_lo, nlz_hi; // Note: nlz_lo may be 1 bit wider than needed.

      // Instantiate recursive modules.
      //
      clz_tree #(wlo,wwhi) clo( nlz_lo, a[wlo-1:0] );
      clz_tree #(whi,wwhi) chi( nlz_hi, a[w-1:wlo] );

      // Split the nlz_lo and nlz_hi outputs into "overflow" (MSB) bits,
      // ov_lo and ov_hi, and the remaining bits lz_lo and lz_hi.
      //
      uwire ov_lo, ov_hi;
      uwire [lhi-1:0] lz_lo, lz_hi;
      assign { ov_lo, lz_lo } = nlz_lo;
      assign { ov_hi, lz_hi } = nlz_hi;

      assign nlz = !ov_hi ? { 2'b00, lz_hi } :  // Case 0
                   !ov_lo ? { 2'b01, lz_lo } :  // Case 1
                            { 2'b10, lz_lo };   // Case 2
      // Case 0:
      //   Input to chi has a 1, so just use nlz_hi.
      //   This case occurs when the MSB of nlz_hi is 0.
      //   For this case just set nlz to nlz_hi.
      //
      // Case 1:
      //   Input to chi is all zeros, and  wlo < whi  or  nlz_lo < whi.
      //   For this case set nlz = whi + nlz_lo = { 2'b01, lz_lo }.
      //
      // Case 2:
      //   Input to chi is all zeros, and  nlz_lo == whi.
      //   If this condition is true then ov_lo = 1
      //   For this case set
      //     nlz = whi + nlz_lo = { 2'b1 + ov_lo, lz_lo } = { 2'b10, lz_lo }
```

```
    end

endmodule
```

Problem 2: Run the synthesis program and indicate how your module compares to the behavioral module, `clz`. Indicate which results are expected, and which are not expected, and explain why.

Attention students studying for exams: A good practice problem would be to show the synthesized hardware for these modules.

The behavioral model looks at bits sequentially, starting at the most-significant bit. The hardware as initially inferred would have a chain of multiplexors either selecting `i` if `aa[i]` were `1`, or the prior value of `nlz` otherwise. The `nlz` output would pass through `w` multiplexors, for a delay of $w\,\mathrm{u_t}$ after optimizing for the fact that `i` is constant.

In contrast the critical path through the tree modules passes through $\lceil \lg w \rceil$ units, and so that should be faster. In the $0.1\,\mathrm{ns}$ delay target results, shown below, the behavioral model is fastest at $w = 30$ bits and the adder-less `clz_tree` module is only fastest at $w = 32$ bits. At best, `clz_tree` never does poorly when delay is a priority. The behavioral module however is consistently more costly than `clz_tree`.

Module Name	Area	Delay Actual	Delay Target
clz_w30	18540	1.653	10.000 ns
clz_tree_w30	17977	1.653	10.000 ns
clz_tree_fat_w30	17977	1.653	10.000 ns
clz_w32	26290	3.110	10.000 ns
clz_tree_w32	21706	1.425	10.000 ns
clz_tree_fat_w32	21401	1.296	10.000 ns
clz_w35	23140	1.300	10.000 ns
clz_tree_w35	22578	1.300	10.000 ns
clz_tree_fat_w35	26073	2.094	10.000 ns
clz_w30_1	30053	0.504	0.100 ns
clz_tree_w30_4	38532	0.650	0.100 ns
clz_tree_fat_w30_1	37798	0.861	0.100 ns
clz_w32_1	36476	1.007	0.100 ns
clz_tree_w32_5	37356	0.577	0.100 ns
clz_tree_fat_w32_2	32254	0.634	0.100 ns
clz_w35_1	37008	0.606	0.100 ns
clz_tree_w35_6	37008	0.606	0.100 ns
clz_tree_fat_w35_1	37008	0.606	0.100 ns

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2019 Homework 2 -- SOLUTION
//

 /// Assignment https://www.ece.lsu.edu/koppel/v/2019/hw02.pdf
 /// Solution Problem 2: https://www.ece.lsu.edu/koppel/v/2019/hw02_sol.pdf


`default_nettype none

////////////////////////////////////////////////////////////////////////////////
///  Problem 1
//
 ///   Complete clz_tree so that it computes the clz of its input recursively.
//
//      [✔] Split the input between two recursive instantiations ..
//          .. and properly combine the results.
//      [✔] Don't forget the terminal case, maybe for w == 1.
//      [✔] For maximum credit, avoid any use of adders ..
//          .. by making the width of the "hi" module a power of 2.
//
//      [✔] Make sure that port connections are the correct size ..
//          .. mismatched ports are Verilog errors in this assignment.
//      [✔] Do not make port widths larger than needed.
//      [✔] Make sure that the testbench does not report errors.
//      [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//
//      [✔] As always, avoid costly, slow, and confusing code.

 /// Solution Discussion
//
// Two solutions appear below.
//
// The clz_tree_fat module is much simpler, but it uses an adder.
// Two points would be deducted if this were given as a solution
//
// The clz_tree module does not use an adder. It would get full credit.


 /// SOLUTION -- With Adder. Two points would be deducted.
module clz_tree_fat
  #( int w = 19,
     int ww = $clog2(w+1) )
   ( output uwire [ww:1] nlz,
     input uwire [w:1] a );

   if ( w == 1 ) begin

      assign nlz = ~ a;

   end else begin
```

```
      localparam int wlo = w/2;
      localparam int whi = w - wlo;
      localparam int wwlo = $clog2(wlo+1);
      localparam int wwhi = $clog2(whi+1);

      uwire [wwlo:1] lz_lo;
      uwire [wwhi:1] lz_hi;

      clz_tree_fat #(wlo) clo( lz_lo, a[wlo:1]   );
      clz_tree_fat #(whi) chi( lz_hi, a[w:wlo+1] );

      assign nlz = lz_hi < whi ? lz_hi : whi + lz_lo;

   end

endmodule


 /// SOLUTION -- Without Adder
module clz_tree
  #( int w = 19,
     int ww = $clog2(w+1) )
   ( output uwire [ww-1:0] nlz,
     input uwire [w-1:0] a );

   if ( w == 1 ) begin

      assign nlz = !a[0];

   end else if ( w == 2 ) begin

      assign nlz = { !a[0] && !a[1], !a[1] && a[0] };

   end else begin

      // Set whi to the largest power of 2 strictly less than w.
      //
      localparam int lhi = $clog2(w) - 1;
      localparam int whi = 1 << lhi;
      localparam int wwhi = lhi + 1;

      // Then set wlo to the number of remaining bits.
      //
      localparam int wlo = w - whi;

      uwire [wwhi-1:0] nlz_lo, nlz_hi;
      // Note: nlz_lo may be one bit wider than needed.

      // Instantiate recursive modules.
      //
      clz_tree #(wlo,wwhi) clz_lo( nlz_lo, a[wlo-1:0] );
      clz_tree #(whi,wwhi) clz_hi( nlz_hi, a[w-1:wlo] );

      // Split the nlz_lo and nlz_hi outputs into "overflow" (MSB) bits,
```

```
      // ov_lo and ov_hi, and the remaining bits lz_lo and lz_hi.
      //
      uwire ov_lo, ov_hi;
      uwire [lhi-1:0] lz_lo, lz_hi;
      //
      assign { ov_lo, lz_lo } = nlz_lo;
      assign { ov_hi, lz_hi } = nlz_hi;

      // Assemble nlz in one of three ways (Case 0, Case 1, Case 1)
      //
      assign nlz = !ov_hi ? { 2'b00, lz_hi } :  // Case 0
                   !ov_lo ? { 2'b01, lz_lo } :  // Case 1
                            { 2'b10, lz_lo };   // Case 2
      //
      // Case 0:
      //   Input to clz_hi has a 1, so just use nlz_hi.
      //   This case occurs when the MSB of nlz_hi is 0.
      //   For this case just set nlz to nlz_hi.
      //
      // Case 1:
      //   Input to clz_hi is all zeros, and  wlo < whi  or  nlz_lo < whi.
      //   For this case set nlz = whi + nlz_lo = { 2'b01, lz_lo }.
      //
      // Case 2:
      //   Input to clz_hi is all zeros, and  nlz_lo == whi.
      //   If this condition is true then ov_lo = 1
      //   For this case set
      //     nlz = whi + nlz_lo = { 2'b1 + ov_lo, lz_lo } = { 2'b10, lz_lo }
   end

endmodule


 /// A Behavioral CLZ Description
module clz
  #( int w = 19,
     int ww = $clog2(w+1) )
   ( output var logic [ww-1:0] nlz,
     input uwire logic [w-1:0] a );

   uwire [w:0] aa = { a, 1'b1 };
   always_comb for ( int i=0; i<=w; i++ ) if ( aa[i] ) nlz = w-i;

endmodule

////////////////////////////////////////////////////////////////////////////////
/// Testbench Code

// cadence translate_off

module testbench;

   // The widths (values of w) at which the modules will be instantiated.
   //
```

```
   localparam int widths[] = { 1, 2, 5, 8, 13, 15, 17 };

   //  localparam int nw = widths.size();
   localparam int nw = 7; // Cadence, please fix this.
   initial if ( nw != widths.size() )
     $fatal(1,"Constant nw should be %0d.\n",widths.size() );

   int t_errs;     // Total number of errors.
   initial t_errs = 0;
   final $write("Total number of errors: %0d\n",t_errs);

   uwire d[nw:-1];    // Start / Done signals.
   assign d[-1] = 1;  // Initialize first at true.

   // Instantiate a testbench at each size.
   //
   for ( genvar i=0; i<nw; i++ )
     testbench_n #(widths[i]) t2( .done(d[i]), .start(d[i-1]) );

endmodule

module testbench_n
  #( int w = 20 )
   ( output logic done, input uwire start );

   localparam int ww = $clog2(w+1);

   localparam int n_tests = w * 10;

   uwire [ww:1] nlz;
   logic [w-1:0] a;
   clz_tree #(w) c0(nlz,a);
   //  clz #(w) c0(nlz,a);

   initial begin

      automatic int n_errs = 0;

      wait( start );

      $write("** Starting tests for width %0d.\n",w);

      for ( int t=0; t<n_tests; t++ ) begin

         automatic int lz = {t} % ( w + 1 );
         a = { 1'b1, (w)'({$random}) } >> ( lz + 1 );

         #1;

         if ( nlz !== lz ) begin
            n_errs++; testbench.t_errs++;
            if ( testbench.t_errs < 5 || n_errs < 2 )
              $write("Error for width %2d: input %h:  %d != %0d (correct).\n",
                     w, a, nlz, lz);
```

```
            end
        end
        $write("Width %0d, done with %0d tests, %0d errors.\n",w,n_tests,n_errs);
        done = 1;
    end
endmodule
// cadence translate_on
```

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2019 Homework 2 -- SOLUTION
//

 /// Assignment https://www.ece.lsu.edu/koppel/v/2019/hw02.pdf
 /// Solution Problem 2: https://www.ece.lsu.edu/koppel/v/2019/hw02_sol.pdf


`default_nettype none

////////////////////////////////////////////////////////////////////////////////
/// Problem 1
//
 ///  Complete clz_tree so that it computes the clz of its input recursively.
//
//     [✔] Split the input between two recursive instantiations ..
//         .. and properly combine the results.
//     [✔] Don't forget the terminal case, maybe for w == 1.
//     [✔] For maximum credit, avoid any use of adders ..
//         .. by making the width of the "hi" module a power of 2.
//
//     [✔] Make sure that port connections are the correct size ..
//         .. mismatched ports are Verilog errors in this assignment.
//     [✔] Do not make port widths larger than needed.
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//
//     [✔] As always, avoid costly, slow, and confusing code.

 /// Solution Discussion
//
// Two solutions appear below.
//
// The clz_tree_fat module is much simpler, but it uses an adder.
// Two points would be deducted if this were given as a solution
//
// The clz_tree module does not use an adder. It would get full credit.


 /// SOLUTION -- With Adder. Two points would be deducted.
module clz_tree_fat
  #( int w = 19,
     int ww = $clog2(w+1) )
   ( output uwire [ww:1] nlz,
     input uwire [w:1] a );

   if ( w == 1 ) begin

      assign nlz = ~ a;

   end else begin
```

```
      localparam int wlo = w/2;
      localparam int whi = w - wlo;
      localparam int wwlo = $clog2(wlo+1);
      localparam int wwhi = $clog2(whi+1);

      uwire [wwlo:1] lz_lo;
      uwire [wwhi:1] lz_hi;

      clz_tree_fat #(wlo) clo( lz_lo, a[wlo:1]   );
      clz_tree_fat #(whi) chi( lz_hi, a[w:wlo+1] );

      assign nlz = lz_hi < whi ? lz_hi : whi + lz_lo;

   end

endmodule


 /// SOLUTION -- Without Adder
module clz_tree
  #( int w = 19,
     int ww = $clog2(w+1) )
   ( output uwire [ww-1:0] nlz,
     input uwire [w-1:0] a );

   if ( w == 1 ) begin

      assign nlz = !a[0];

   end else if ( w == 2 ) begin

      assign nlz = { !a[0] && !a[1], !a[1] && a[0] };

   end else begin

      // Set whi to the largest power of 2 strictly less than w.
      //
      localparam int lhi = $clog2(w) - 1;
      localparam int whi = 1 << lhi;
      localparam int wwhi = lhi + 1;

      // Then set wlo to the number of remaining bits.
      //
      localparam int wlo = w - whi;

      uwire [wwhi-1:0] nlz_lo, nlz_hi;
      // Note: nlz_lo may be one bit wider than needed.

      // Instantiate recursive modules.
      //
      clz_tree #(wlo,wwhi) clz_lo( nlz_lo, a[wlo-1:0] );
      clz_tree #(whi,wwhi) clz_hi( nlz_hi, a[w-1:wlo] );

      // Split the nlz_lo and nlz_hi outputs into "overflow" (MSB) bits,
```

```
      // ov_lo and ov_hi, and the remaining bits lz_lo and lz_hi.
      //
      uwire ov_lo, ov_hi;
      uwire [lhi-1:0] lz_lo, lz_hi;
      //
      assign { ov_lo, lz_lo } = nlz_lo;
      assign { ov_hi, lz_hi } = nlz_hi;

      // Assemble nlz in one of three ways (Case 0, Case 1, Case 1)
      //
      assign nlz = !ov_hi ? { 2'b00, lz_hi } :  // Case 0
                   !ov_lo ? { 2'b01, lz_lo } :  // Case 1
                            { 2'b10, lz_lo };   // Case 2
      //
      // Case 0:
      //   Input to clz_hi has a 1, so just use nlz_hi.
      //   This case occurs when the MSB of nlz_hi is 0.
      //   For this case just set nlz to nlz_hi.
      //
      // Case 1:
      //   Input to clz_hi is all zeros, and  wlo < whi  or  nlz_lo < whi.
      //   For this case set nlz = whi + nlz_lo = { 2'b01, lz_lo }.
      //
      // Case 2:
      //   Input to clz_hi is all zeros, and  nlz_lo == whi.
      //   If this condition is true then ov_lo = 1
      //   For this case set
      //     nlz = whi + nlz_lo = { 2'b1 + ov_lo, lz_lo } = { 2'b10, lz_lo }
   end

endmodule


 /// A Behavioral CLZ Description
module clz
  #( int w = 19,
     int ww = $clog2(w+1) )
   ( output var logic [ww-1:0] nlz,
     input uwire logic [w-1:0] a );

   uwire [w:0] aa = { a, 1'b1 };
   always_comb for ( int i=0; i<=w; i++ ) if ( aa[i] ) nlz = w-i;

endmodule

////////////////////////////////////////////////////////////////////////////////
/// Testbench Code

// cadence translate_off

module testbench;

   // The widths (values of w) at which the modules will be instantiated.
   //
```

```
   localparam int widths[] = { 1, 2, 5, 8, 13, 15, 17 };

   //  localparam int nw = widths.size();
   localparam int nw = 7; // Cadence, please fix this.
   initial if ( nw != widths.size() )
     $fatal(1,"Constant nw should be %0d.\n",widths.size() );

   int t_errs;     // Total number of errors.
   initial t_errs = 0;
   final $write("Total number of errors: %0d\n",t_errs);

   uwire d[nw:-1];    // Start / Done signals.
   assign d[-1] = 1;  // Initialize first at true.

   // Instantiate a testbench at each size.
   //
   for ( genvar i=0; i<nw; i++ )
     testbench_n #(widths[i]) t2( .done(d[i]), .start(d[i-1]) );

endmodule

module testbench_n
  #( int w = 20 )
   ( output logic done, input uwire start );

   localparam int ww = $clog2(w+1);

   localparam int n_tests = w * 10;

   uwire [ww:1] nlz;
   logic [w-1:0] a;
   clz_tree #(w) c0(nlz,a);
   //  clz #(w) c0(nlz,a);

   initial begin

      automatic int n_errs = 0;

      wait( start );

      $write("** Starting tests for width %0d.\n",w);

      for ( int t=0; t<n_tests; t++ ) begin

         automatic int lz = {t} % ( w + 1 );
         a = { 1'b1, (w)'({$random}) } >> ( lz + 1 );

         #1;

         if ( nlz !== lz ) begin
            n_errs++; testbench.t_errs++;
            if ( testbench.t_errs < 5 || n_errs < 2 )
              $write("Error for width %2d: input %h:  %d != %0d (correct).\n",
                     w, a, nlz, lz);
```

```
            end
        end
        $write("Width %0d, done with %0d tests, %0d errors.\n",w,n_tests,n_errs);
        done = 1;
    end
endmodule
// cadence translate_on
```

LSU EE 4755 Homework 3 Solution Due: 23 October 2019

Problem 1: Appearing below is a module excerpted from the solution to Homework 1. Compute the cost and delay of this module using the simple model under the following assumptions:

- The inputs arrive at $t = 0$. Don't assume that any bit is early or late, they all arrive at exactly $t = 0$.
- A ripple adder will be used to implement addition.
- Apply obvious optimizations. In particular, don't use a BFA if a BHA would suffice. And only use a BHA if that is needed.
- Don't overlook the fact that one of the shifter inputs is a constant.

Show the cost and delay in terms of `wa` and `wb`, but use symbol a for `wa` and b for `wb`. For example, "The cost is $(a+b)9\,\mathrm{u_c}$ and the delay is $(a+b)2\,\mathrm{u_t}$." (Those answers assume that BFAs are used for the entire module, which is wrong.)

The simple model slides (AOTW) don't show the cost and delay of a BHA, so work that out yourselves.

```
module mult_piece
  #( int wa = 16, int wb = 16, int wp = wa + wb,
     int wn = wa / 2, int wx = wb + wn )
   ( output uwire [wp:1] prod,
     input uwire [wx:1] prod_lo, prod_hi );

      assign prod = prod_lo + ( prod_hi << wn );

endmodule
```

Short answer: Cost: $[9b + 3\frac{a}{2}]\,\mathrm{u_c}$, Delay: $[2 + 2b + \frac{a}{2}]\,\mathrm{u_t}$.

Long answer: Because `prod_hi` is shifted and because `prod_lo` and `prod_hi` are the same width the adder can be broken into three regions: an $a/2$-bit low region consisting of the low $a/2$ bits of `prod_lo`, a b-bit middle region consisting of the high b bits of `prod_lo` and the low b bits of `prod_hi`, and an $a/2$-bit region consisting of the high $a/2$ bits of `prod_hi`.

There is no hardware at all for the low region. The middle region consists of b binary full adders, and the high region consists of $a/2$ binary half adders. (The high region has to handle the carry out from the middle region.)

Under the simple model a BFA cost $9\,\mathrm{u_c}$ and in a w-bit ripple configuration has a delay of $[2 + 2w]\,\mathrm{u_t}$. A BHA can be derived from a BFA by setting the `a` or `b` input to logic zero and then simplifying. Such a BHA would have a cost of $3\,\mathrm{u_c}$ per bit and a delay of $1\,\mathrm{u_t}$ per bit in a ripple configuration.

The total cost is then $[9b + 3\frac{a}{2}]\,\mathrm{u_c}$ and the delay is $[2 + 2b + \frac{a}{2}]\,\mathrm{u_t}$.

There's another problem on the next page!

Problem 2: A w-bit multiplier needs to add together w partial products using $w-1$ adders. A naïve timing analysis of a non-tree ripple adder implementation would compute a delay of $w(2 \times 2w+2) = (4w^2+w)\,\mathrm{u_t}$ for the $2w$-bit product using the simple model and ignoring ripple-unit cascading. As we should know $4w^2$ is not a good term to have in an expression for time. The goal of this problem is to see how the tree multiplier compares to this naïve timing analysis.

Appearing below is the Bonus Solution to Homework 1 in which a single `mult_tree` module is used rather than separate modules `mult16_tree`, `mult8_tree`, etc. Also shown is a module, `my_module`, that instantiates the `mult_tree`. Also shown a page or two ahead is the diagram from Homework 1. You may want to use this to help work out the solution to this problem.

Analyze the cost and performance of `my_module` as described below. When computing the cost and performance don't forget to account for the full elaboration, not just the top level. For example, `my_module` with `w=4` consists of one `mult_tree` at `w=4` and two `mult_tree` modules at `w=2`, and four `mult_tree` modules at `w=1`.

```
module mult_tree
  #( int wa = 16, int wb = 16, int wp = wa + wb )
   ( output logic [wp:1] prod,
     input uwire [wa:1] a,
     input uwire [wb:1] b );

   if ( wa == 1 ) begin

      assign prod = a ? b : 0;
      // Equivalent to: prod = a * b;

   end else begin

      // Split a in half and recursively instantiate a module for each half.
      localparam int wn = wa / 2;
      localparam int wx = wb + wn;

      uwire [wx:1] prod_lo, prod_hi;

      mult_tree #(wn,wb) mlo( prod_lo, a[wn:1],    b );
      mult_tree #(wn,wb) mhi( prod_hi, a[wa:wn+1], b );

      // Combine the partial products.
      always_comb prod = prod_lo + ( prod_hi << wn );

   end
endmodule

module my_module
  #( int w = 8, int wp = 2 * w )
   ( output uwire [wp-1:0] p,
     input uwire [w-1:0] x, y );
   mult_tree #(w,w) mt1(p,x,y);
endmodule
```

(*a*) Compute the cost of `my_module` using the same assumptions as in Problem 1. The cost must

be in terms of w. It's okay, indeed encouraged, to use sample values like $w = 16$ when working out the problem, but once you have it figured out give the answer in terms of w. (If you have not solved Problem 1 then use the incorrect sample answers provided in Problem 1.)

The following identity may be helpful: $\sum_{i=0}^{m-1} 2^i = 2^m - 1$. In such a summation i might indicate the level of recursion and 2^i might indicate the number of modules at that recursion level. For the top level of the recursion $i = 0$.

Let j denote the recursion level such that $a = 2^j$, and note that j starts at $\lg w$ with the initial instantiation of `mult-tree` (the one made by `my-module`) and ends at $j = 0$, the terminal case. At level j there are a total of $w/2^j$ instances. For the terminal case, $j = 0$ and $a = 1$, `mult-tree` produces just a mux, which itself will be optimized to b AND gates. There will be $w/2^0$ instances for $j = 0$, so their total cost will be $w^2\,\mathrm{u_c}$, after setting $b = w$.

The $j > 0$ levels consist of binary full and half adders. Each instance has about w BFAs and $a/2 = 2^{j-1}$ BHAs. Let c_f denote the per-bit cost of a BFA and c_h denote the per-bit cost of a BHA. By the simple model $c_f = 9\,\mathrm{u_c}$ and $c_h = 3\,\mathrm{u_t}$. (In the BHA the carry out can be used to compute the sum, reducing the number of additional gates for the XOR to 2.) Then the total cost of the adders is

$$\begin{aligned}
&\sum_{j=1}^{\lg w} \frac{w}{2^j}\left(wc_f + 2^{j-1}c_h\right) \\
=&w\sum_{j=1}^{\lg w}\left(\frac{wc_f}{2^j} + \frac{c_h}{2}\right) \\
=&w^2\sum_{j=1}^{\lg w}\frac{c_f}{2^j} + \frac{w}{2}\sum_{j=1}^{\lg w} c_h \\
=&w^2\left(1 - \frac{1}{w}\right)c_f + \frac{w}{2}(\lg w)c_h
\end{aligned}$$

(Note that $\sum_{j=1}^{\lg w} 2^{-j} = (1 - 1/w)$.) The total overall cost is

$$\begin{aligned}
&w^2\,\mathrm{u_c} + w^2\left(1 - \frac{1}{w}\right)c_f + \frac{w}{2}(\lg w)c_h \\
=&\left[10w^2 - 9w + \frac{3w}{2}(\lg w)\right]\,\mathrm{u_c}
\end{aligned}$$

An important point to note is that the cost is proportional to w^2. That should not be surprising because we know that to multiply two w-bit quantities we need $w - 1$ adders, each costing about wc_f.

(b) Compute the delay of the multiplier using a simplifying assumption similar to the one used in Problem 1: when computing the delay of `prod = prod_lo + ( prod_hi << wn )` assume that all bits for `prod_lo` and `prod_hi` arrive at the same time and that all bits of `prod` are sent to the outputs at the same time. (Don't like simplifying assumptions? The next subproblem is for you!)

Show your answer for `w=8` and as an expression in terms of w. Don't forget to consider the entire elaboration, not just the top-level module.

The launch point starts at $j = 0$ (the terminal case), which depends only on the inputs to `my-module`, `a` and `b`. The delay is just $1\,\mathrm{u_t}$.

Level $j > 0$ has an adder consisting of w BFAs and 2^{j-1} BHAs. The total delay through that is $[2(w-1) + 2^{j-1}]\,\mathrm{u_t}$, where the delay through a 2^{j-1}-bit BHA is $2^{j-1}\,\mathrm{u_t}$. The total delay including the AND gates is

$$\begin{aligned}
&\left[1 + \sum_{j=1}^{\lg w} \left(2(w-1) + 2^{j-1}\right)\right] \mathrm{u_t} \\
&= \left[1 + 2(w-1)(\lg w) + w - 1\right] \mathrm{u_t} \\
&= \left[w + 2(w-1)(\lg w)\right] \mathrm{u_t} \\
&\approx 2w \lg w \, \mathrm{u_t}
\end{aligned}$$

The dominant term is $2w \lg w$ which is not as bad as a linear connection of adders which would have a delay of $\approx 2w^2 \, \mathrm{u_t}$ under similar assumptions.

(*c*) Compute the delay of the multiplier without the simplifying assumption. That is, account for the fact that the less-significant bits of `mult_tree` will be ready before the more-significant bits.

Show your answer for `w=8` and as an expression in terms of w. Don't forget to consider the entire elaboration, not just the top-level module.

At level j the least significant BFA (which could actually be a BHA, but we'll keep it simple) is connected to bit 2^{j-1} of `prod-lo` and bit zero of `prod-hi`. Since level j is waiting for 2^{j-1} to be ready, the next level, $j+1$, must be waiting for bit 2^j. Therefore for level j we need to compute the delay though $2^{j-1}+1$ BFAs: stating at the least significant BFA (bit position 2^{j-1}) and ending at the BFA computing bit 2^j. The delay for w bits for a ripple adder under the simple model is $2(w+1) \, \mathrm{u_t}$, so the delay at level j before $j+1$ can start is $2((2^{j-1}+1)+1) = 2^j + 4$.

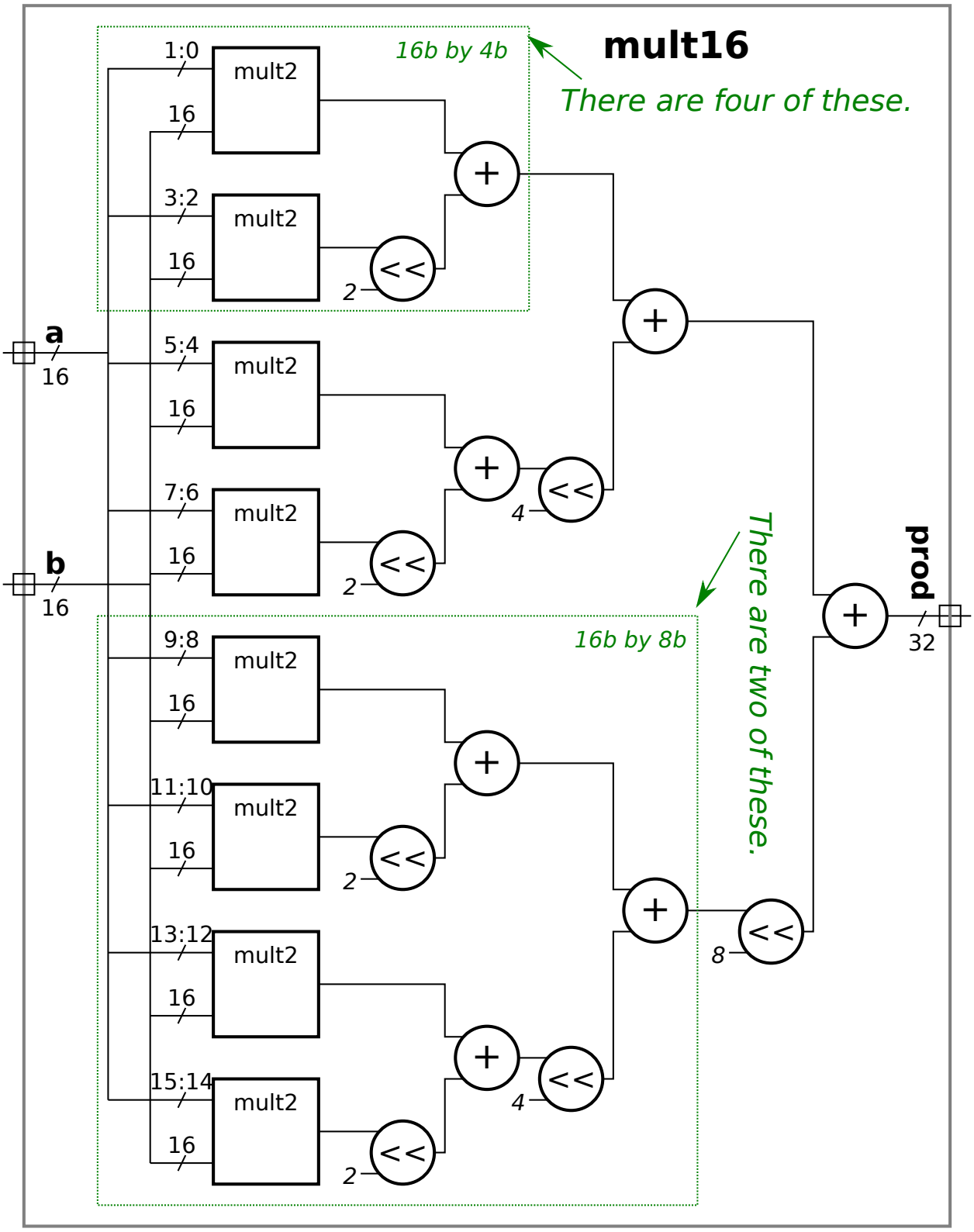
For level $j = 0$ the delay is $1 \, \mathrm{u_t}$ (an AND gate). For level $j = \lg w$ we need to add on the remaining bits in the ripple adder: $w/2 - 1$ BFA delays and $w/2$ BHA delays.

The total delay is:

$$\begin{aligned}
&\left[1 + \left(\sum_{j=1}^{\lg w} 2^j + 4\right) + \left(\frac{w}{2} - 1\right) 2 + \frac{w}{2}\right] \mathrm{u_t} \\
&= \left[1 + 2(2w-1) + 4(\lg w) + \left(\frac{w}{2} - 1\right) 2 + \frac{w}{2}\right] \mathrm{u_t} \\
&= \left[5.5w + 4(\lg w) - 1\right] \mathrm{u_t}
\end{aligned}$$

Useful diagram on next page.

Use the diagram below to help work out solutions.



LSU EE 4755 Homework 4 Solution Due: 11 November 2019

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/2019/hw04.v.html`.

Problem 0: Following instructions at `https://www.ece.lsu.edu/koppel/v/proc.html`, set up your class account (if for whatever reason you haven't done so or neeed to do it again), copy the assignment, and run the Verilog simulator and synthesis program on the unmodified homework file, `hw04.v`. Do this early enough so that minor problems (*e.g.*, password doesn't work) are minor problems.

Homework Overview

Module `best_match_behavioral` has two inputs, a longer vector, `val`, and a short vector, `k`. It sets `pos` to the start of a subvector of `val` that best matches `k` and sets `err` to the number of bit positions that don't match. For example, suppose `val = 8'b11110000` and `k=4'b1100`. Then `pos` would be set to 2 and `err` to 0 because there is an exact match at position 2 in `val`. If `k=4'b1101` then there isn't an exact match for `k` in `val`, but at position 2 there is a match with one error. If `k=2'b00` then there are matches at positions 0, 1, and 2, all with zero errors.

Module `best_match_behavioral` is combinational (and was written as a behavioral module). In this assignment a sequential version will be written and analyzed.

Testbench Code

The testbench for this assignment, which can be run when visiting the file in Emacs in a properly set-up account by pressing `F9`, tests the modules. Initially, the testbench will exit because module `best_match` has not responded in sufficient time. When that happens one of the last lines of the testbench output shows that the final cycle count is the same as the cycle limit (128 below), and "CYCLE LIMIT EXCEEDED" is shown.

```
ncsim> run
Exit from clock loop at cycle 128, limit 128.  ** CYCLE LIMIT EXCEEDED **
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit

Compilation finished at Mon Nov  4 17:56:24
```

To get rid of this message `best_match` must handshake correctly, see Problem 1. If `best_match` responds in time, the testbench will check to see if `pos` is in the right range. The output below shows errors when `pos` is out of range: `Error in best_match, test #     3, pos out of range: 0xff`

```
Error in best_match, test #     4, pos out of range: 0xff
Done with best_match_behavioral tests,           0 errors found.
Done with best_match tests,        1000 errors found.
Exit from clock loop at cycle 59001, limit 59069.
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
```

The output `err` is supposed to be the number of non-matching bits at `pos`. If not, the testbench shows output like:

```
Error in best_match, test #     2, err wrong 1 != 3 (correct) pos   2  84 ^ 01
Error in best_match, test #     3, err wrong 1 != 2 (correct) pos  13  1f ^ 3d
Error in best_match, test #     4, err wrong 1 != 2 (correct) pos   4  78 ^ f9
Done with best_match_behavioral tests,           0 errors found.
```

```
Done with best_match tests,         972 errors found.
Exit from clock loop at cycle 59001, limit 59069.
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
```

For test # 4, the testbench reports that `err` was 1 but should have been 2. The line also shows that `pos` was set to 4, and that `val` at that position was `78` (in hexadecimal) and that `k=f9`.

The testbench also checks whether the `err` returned is the minimum error for that value of `val` and `k`.

The testbench prints the details of the first few errors it finds. A grand total is printed at the end, see the transcript above.

Use Simvision to debug your modules. Feel free to modify the testbench so that it presents inputs that facilitate debugging.

Synthesis

The synthesis script, `syn.tcl`, will synthesize `best_match_behavioral` (for reference) and `best_match` (your solution). Each module will be synthesized at three widths, and with two delay targets, an easy 90 ns and a un-achievable 0.1 ns. If a module doesn't synthesize −.001 s is shown for its delay. The script is run using the shell command `genus -files syn.tcl`, which invokes Cadence Genus. If you would like to synthesize additional modules or sizes edit `syn.tcl` near the bottom.

The synthesis script shows area (cost), delay, and the delay target in a neat table. Additional output of the synthesis program is written to file `spew-file.log`.

Problem 1 on next page.

Problem 1: Complete module `best_match` so that it computes the best match sequentially as described below. In addition to `val` and `k`, the module has 1-bit inputs `start` and `clk` and 1-bit output `ready`.

Handshaking works as follows: When `start=1` at a positive edge the module should set `ready` to zero. It should then start scanning for the best match, checking one shifted position per cycle. The maximum number of cycles needed should be `wv-wk` plus one or two more needed for handshaking. (The testbench will wait `2*wv` cycles before giving up.) The module should set `err` and `pos` to their correct values and `ready` to 1.

The inputs, `val` and `k` will be held steady at least until `ready` is set to 1.

The module must use the `pop` (population) module (in `hw04.v`) to compute possible values for `err`. That is, don't use something like the `b` loop in `best_match_behavioral` to accumulate the sum `e`. Instead compute the XOR of the appropriate bit range and provide that to the `pop` module as an input.

For maximum credit avoid the use of large (such as `wv`-input) multiplexors in your design, or the use of a non-constant shifter.

The module must be synthesizable and correct.

The behavioral best match module is shown below for reference.

```
module best_match_behavioral
  #( int wv = 32, int wk = 10, int wvb = $clog2(wv), int wkb = $clog2(wk+1) )
   ( output logic [wvb:1] pos,  // Position of best match.
     output logic [wkb:1] err,  // Number of non-matching bits.
     input uwire [wv-1:0] val, input uwire [wk-1:0] k );

   always_comb begin

      automatic int best_err = wk + 1;
      automatic int best_pos = -1;

      for ( int p=0; p<=wv-wk; p++ ) begin
         automatic int e = 0;
         for ( int b=0; b<wk; b++ ) e += k[b] !== val[p+b];
         if ( e < best_err ) begin
            best_err = e;
            best_pos = p;
         end
      end
      err = best_err;
      pos = best_pos;

   end

endmodule
```

Solution on next page.

The solution appears below. The biggest difference between **best_match_behavioral** and **best_match** is that the **p**-loop has been eliminated, and the iteration variable, **p**, has been declared as a variable. The variable **p** is initialized to zero when **start** is asserted and then incremented each cycle until it points to the last position of a possible match, **wv-wp**.

Another difference is that the **b** loop, used to total the number of incorrect bit positions, has been replaced by the **pop** module. The input to the **pop** module is a bit vector, **e_vec**, which is constructed by exclusive-or'ing **k** with the low bits of **sh_val**. Bit **i** of **e_vec** is 1 if the bit **i** of **k** is different than bit **i** of **sh_val**, bit **i** is 0 if the bits are the same. Equivalently, **e_vec[i] = k[i] !== sh_val[i]**, or **e_vec[i] = k[i] ^ sh_val[i]**. Rather than iterating over **i** the entire value is computed using the bitwise exclusive OR operator: **e_vec = k ^ sh_val**.

The register **sh_val** is initialized to **val** and then shifted right by one bit each iteration. This avoids the need for a shifter. For example, if the error vector were computed using **e_vec = k ^ val[p +: wk];** a shifter would be needed for **val[p +: wk]**, to extract **wk** bits starting at position **p**.

```
module best_match
  #( int wv = 32, int wk = 10, int wvb = $clog2(wv), int wkb = $clog2(wk+1) )
   ( output logic [wvb:1] pos,  output logic [wkb:1] err,  output logic ready,
     input uwire [wv-1:0] val,  input uwire [wk-1:0] k,    input uwire start, clk );

   logic [wv-1:0] sh_val;
   logic [wvb-1:0] p;

   uwire [wk-1:0] e_vec = k ^ sh_val[wk-1:0];
   uwire [wkb-1:0] e;
   pop #(wk,wkb) p1( e, e_vec );

   always_ff @( posedge clk )

     if ( start == 1 ) begin

        ready = 0;
        sh_val = val;
        p = 0;
        err = wk;  // wk+1 might overflow err.

     end else if ( !ready ) begin

        if ( e < err ) begin  err = e;  pos = p;  end

        ready = p == wv - wk;
        p++;
        sh_val >>= 1;
     end

endmodule
```

Problem 2: Run the synthesis program and indicate how your module compares to the behavioral module.

Synthesis results appear below.

Module Name	Area	Delay Actual	Delay Target
best_match_wv16	47923	3.786	90.000 ns
best_match_mux_wv16	46566	5.181	90.000 ns
best_match_behavioral_wv16	87155	10.862	90.000 ns
best_match_wv24	60757	3.675	90.000 ns
best_match_mux_wv24	60566	5.503	90.000 ns
best_match_behavioral_wv24	192546	21.535	90.000 ns
best_match_wv16_2	63287	2.413	0.100 ns
best_match_mux_wv16_3	77134	3.398	0.100 ns
best_match_behavioral_wv16_137	231102	3.504	0.100 ns
best_match_wv24_1	79273	2.652	0.100 ns
best_match_mux_wv24_2	89081	3.590	0.100 ns
best_match_behavioral_wv24_297	563769	6.667	0.100 ns

(*a*) Compare the amount of time needed for your module compared to the behavioral one. The answer to this question requires some manipulation of the values in the `Delay Actual` column. Indicate which results are expected, and which are not expected, and explain why.

The manipulation alluded to in the question is the multiplying of the delay by the number of cycles needed to compute the result (the position and error of the best match). The behavioral module is combinational, and so only one "cycle" is needed. (It's not really a cycle because the module isn't clocked.) The `best_match` module, in contrast, requires `wv-wk` cycles and so the delay must be multiplied by that number of cycles, $24-10=14$ for the 24-bit module and $16-10=6$ for the 16-bit module, in order to compute the time needed to find the best match.

Since we are comparing time we should look at the results for a delay target of $0.1\,\mathrm{ns}$ because it is in those runs that the synthesis program is optimizing delay. For the 24-bit module the behavioral module requires $6.667\,\mathrm{ns}$ to compute the best match. For `wv=24` and `wk=10` the `best_match` module in this solution requires at least $24-10=14$ cycles, for a total time of $14\times 2.652\,\mathrm{ns}=37.128\,\mathrm{ns}$. So the behavioral module will compute the error and position in much less time.

Some students submitted solutions that used fewer than `wv-wk` cycles when a perfect match (`err==0`) was found before bit `wv-wk` was reached. A student eager to showcase this clever shortcut could answer this question by describing a favorable situation: "For situations in which a perfect match occurs half the time and is uniformly distributed, ...

The question also asks for a discussion of whether the synthesis delay results were expected. That means we need to make some kind of a delay estimate for each module and compare it to the delays provided by the synthesis program. A starting point for the delay comparison is to recognize a key difference between the two modules: the behavioral module computes the error for each of $24-10=14$ positions (for the 24-bit module) in one cycle while the `best_match` module computes just one position per cycle. This factor of 14 (or $v-k$) difference would seem to put the behavioral module at a disadvantage. An important question to answer is whether the behavioral module's delay should be 14 times larger, $\lceil \lg 14 \rceil$ times larger, or something else. In the module generated by the synthesis program the behavioral delay is $6.667/2.652\approx 2.5$ times larger.

The question did not explicitly ask us to compute the delay (say using the simple model), so that gives us some latitude for approximation. Full credit would have been given for all of the key points made so far in this solution. But having gotten this far, how can we not proceed further into the delay analysis? (Warning: EE 4755 Fall 2019 students are

expected to read the entire solution. Exam questions will be based on homework assignments and the posted solutions, even the excessively wordy ones.)

First, consider the `b` loop in the behavioral module. It is doing the same thing as the assignment to `e_vec` and the hardware in `pop` module are doing in `best_match`. Let's assume that both will be synthesized to the same hardware after optimization (though in the case of the behavioral module there will be `wv-wk` copies of the hardware).

An important thing to remember is that the `p` loop and the `b` loop describe how synthesized hardware will be interconnected. They **do not execute** and don't even exist in the synthesized hardware. (The Verilog simulator does execute the loops as procedural code, but in this part we're considering synthesis.) The expression `k[b] !== val[p+b]` is synthesized into $(v-k)k$ pieces of hardware, one for each possible value of `p` ($(v-k)$ possible values) and `b` (k possible values). Since the values of `k` and `val` are available the beginning of a clock cycle all $(v-k)k$ comparisons are done simultaneously. The `b` loop describes a series of adders, computing the same sum as the `pop` module though describing the sum as a linear sequence of additions. If the synthesis program does its job well, meaning that it can re-associate the linear sequence of additions into a reduction tree, the delay for this will be $2 \lg k$ BFAs. Because of the way the BFAs are connected (possible final exam question?) we'll set the adder delay to 4 per BFA, for a total of $8 \lg k \, \mathrm{u_t}$. The input to each `b` loop is the same `val` and `k`, which are available at the start of a clock cycle. So taking into account the XOR delay each sum will be available at $[2 + 8 \lg k] \, \mathrm{u_t}$.

So far it looks like the time for the behavioral model to compute $v-k$ values of `e` is the same as the time needed by `best_match` to compute one. The difference in timing between the two is due to the code starting at `if ( e < best_err ) begin`. The problem is `best_err`. The value at iteration `p` depends on iteration `p-1` for `p>0`. Variable `best_err` is a live-in and live-out for an iteration. It's critical path passes through a comparison, `e < best_err`, and a mux (selecting the old or new `best_err`). If the comparison has delay $2 \lg k$ and the mux 2, the delay for $v-k$ iterations will be $[2 + 8 \lg k + (2(\lg k) + 2)(v-k)] \, \mathrm{u_t}$. The delay for `best_match` is roughly the of one iteration, $[2 + 8 \lg k + (2(\lg k) + 2)] \, \mathrm{u_t}$.

When $v-k$ is large the behavioral module would take $(v-k)$ times longer based on this analysis. For for $v-k = 24-10 = 14$ and $\lg k = 4$ the delays are much closer. For the behavioral delay $2+8\times4+(2\times8+2)(14) = 316 \, \mathrm{u_t}$ and for `best_match` delay $2 + 8 \times 4 + (2 \times 8 + 2) = 82 \, \mathrm{u_t}$, the modules have less than factor of 4 difference in delay. The synthesis program gives a difference of 2.5. Perhaps the synthesis program used a reduction tree for the `if ( e < best_err )` code. In that case the critical path would be through $\lceil \lg(v-k) \rceil = \lceil \lg 14 \rceil = 4$ layers, in which case delay would be $2 + 8 \times 4 + (2 \times 8 + 2)(4) = 136 \, \mathrm{u_t}$, which works out to $136/82 = 1.66$ times longer than `best_match`. The difference in delays obtained from the synthesis program, 2.5, is somewhere between these two possibilities.

(*b*) Compare the area of your design to the behavioral one. Indicate which results are expected, and which are not expected, and explain why.

For the area comparison the 90 ns delay target runs should be used. For `wv=24` and `wk=10` the `p` loop iterates 14 times and so we would expect the behavioral code to have $14\times$ as much addition (including the `pop` module) and comparison hardware. The `best_match` module though needs a register for `sh_val`, something which the behavioral module does not need. Assume that the `pop` module and the expression totaling `e` use $2k$ BFAs each. (Approximated using $\sum_{i=1}^{\lg k} ik/2^i = 2(k-1) - \lg k$.) At a cost of $9 \, \mathrm{u_c}$ per BFA, the cost of just the adders would be $18k \, \mathrm{u_c}$. For `best_match` there would be only one set of such adders, but for the behavioral module there would be $v-k$ such adders. For $v = 24$ and $k = 10$ the costs would be $180 \, \mathrm{u_c}$ versus $2520 \, \mathrm{u_c}$ for the behavioral module. Using $7 \, \mathrm{u_c}$ per bit for `sh_val`, `pos`, and `err`, `best_match` would also require $7(v + \lg v + \lg(k+1)) = 7(24+5+4) = 231 \, \mathrm{u_c}$ that the behavioral module lacks. The total so far is $411 \, \mathrm{u_c}$ versus $2520 \, \mathrm{u_c}$, a factor of 6 difference. The actual difference is closer to a factor of 3 when optimizing for area.

LSU EE 4755 Homework 5 Solution Due: 20 November 2019

Problem 1: Solve 2018 Final Exam Problem 3, in which the inferred hardware for a `misc` module is to be found (a) and the state of the event queue over time simulating `misc` (b) is to be found.

See the final exam solution at `https://www.ece.lsu.edu/koppel/v/2018/fe_sol.pdf`.

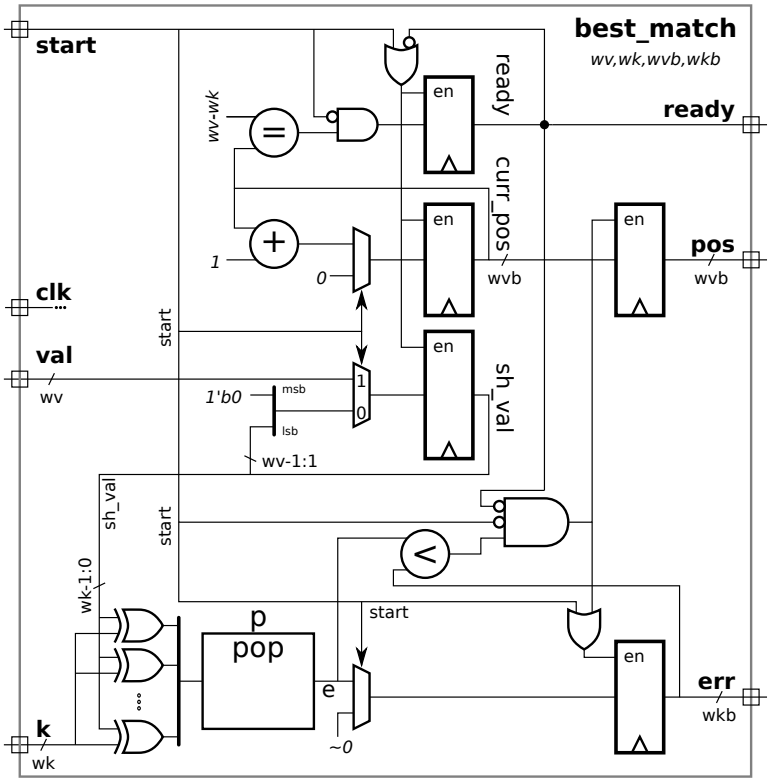
Problem 2 on next page.

Problem 2: Appearing below is a solution to Homework 4 Problem 1. Show the hardware that will be inferred for this module after some optimization. Show the pop module as a box.

- Clearly show all input and output ports.
- Please don't get parameters and ports confused.

Solution appears below. The solution uses enable signals on the registers, but it would also be correct to use an extra mux instead. Because optimization is applied the **wv-wk** term is shown as a constant, not as a subtraction unit. The >>=1 is shown as a bit renumbering instead of has a shift unit.

```
module best_match
  #( int wv = 32, int wk = 10,  int wvb = $clog2(wv),      int wkb = $clog2(wk+1) )
   ( output logic [wvb:1] pos,  output logic [wkb:1] err,  output logic ready,
     input uwire [wv-1:0] val,  input uwire [wk-1:0] k,    input uwire start, clk );

   logic [wvb-1:0] curr_pos;
   logic [wv-1:0] sh_val;
   uwire [wkb-1:0] e;
   pop #(wk,wkb) p( e, k ^ sh_val[wk-1:0] );


   always_ff @( posedge clk )

     if ( start == 1 ) begin

        ready = 0;
        curr_pos = 0;
        sh_val = val;
        err = ~0;

     end else if ( !ready ) begin

        if ( e < err ) begin
          err = e;
          pos = curr_pos;
        end

        ready = curr_pos == wv-wk;
        curr_pos++;
        sh_val >>= 1;

     end

endmodule
```



```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2019 Homework 6 -- SOLUTION
//

 /// Assignment https://www.ece.lsu.edu/koppel/v/2019/hw06.pdf


`default_nettype none

////////////////////////////////////////////////////////////////////////////////
/// Problem 1
//
 ///  Complete add_accum so that it accumulates a sum.
//
//     [✔] Put your solution in add_accum. No other modules should
//         be modified. (Except the testbench, to help debug.)
//
//     [✔] add_accum must use an add_pipe module to compute the sum.
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//
//     [✔] As always, avoid costly, slow, and confusing code.
//     [✔] As always, don't assume parameters will be at their default values.


module add_accum
  #( int w = 21, n_stages = 3 )
   ( output logic [w-1:0] sum,
     output logic sum_valid,
     input uwire [w-1:0] ai,
     input uwire ai_valid, reset, clk );

   /// SOLUTION

   // Register to keep track of which stage of add pipeline is occupied.
   //
   logic [n_stages:0] st_occ;
   //
   // If st_occ[i] == 1 then stage i of pipelined adder is occupied.

   // If none of the adder stages is occupied the value in sum must be valid.
   //
   assign sum_valid = !st_occ;

   // If true, there is a useful result at the adder output.
   //
   uwire aout_valid = st_occ[n_stages-1];

   // Connections to adder.
   //
   uwire [w-1:0] aout;
   uwire [w-1:0] a0 = ai_valid   ? ai   : sum;
   uwire [w-1:0] a1 = aout_valid ? aout : sum;

   add_pipe #(w,n_stages) add_p0( aout, a0, a1, clk );

   // If true, the value in sum is needed.
   //
   logic sum_occupied;

   // Number of values ready to be added together.
   //
   uwire [1:0] n_values = ai_valid + sum_occupied + aout_valid;
   //
   // If n_values == 0: Nothing to do.
   // If n_values == 1: Put or keep the value in sum.
   // If n_values == 2: Put the two values into the adder.
   // If n_values == 3: Put ai and aout into the adder and leave sum unchanged.

   // If true, a pair of values will be put in the adder in this cycle.
```

```
   //
   uwire start_an_addition = n_values >= 2;

   // If true, write sum with either ai or aout.
   //
   uwire write_sum = !sum_occupied && n_values == 1;

   always_ff @( posedge clk ) if ( reset ) begin

      sum <= 0;
      sum_occupied <= 0;
      st_occ <= 0;        // Set occupied bit of every stage to 0.

   end else begin

      if ( write_sum ) sum <= aout_valid ? aout : ai;

      // sum will be occupied if there are an odd number (1 or 3) values ..
      //
      sum_occupied <= n_values[0];
      //
      // .. because if there were 2 values they both would go into the adder.

      // Advance occupied bit by one stage.
      //
      st_occ <= { st_occ[n_stages-1:0], start_an_addition };

   end

endmodule

`ifdef xxx
Synthesizing at effort level "high"

Module Name                              Area   Delay   Delay
                                              Actual  Target
add_pipe_w24_n_stages1                  29928  10.174  90.000 ns
add_pipe_w24_n_stages2                  47043   5.428  90.000 ns
add_pipe_w24_n_stages3                  64159   3.701  90.000 ns
add_pipe_w24_n_stages4                  81275   2.837  90.000 ns
add_pipe_w24_n_stages6                 115506   1.973  90.000 ns

add_accum_w24_n_stages1                 87556  11.449  90.000 ns
add_accum_w24_n_stages2                105305   6.349  90.000 ns
add_accum_w24_n_stages3                123530   4.560  90.000 ns
add_accum_w24_n_stages4                141598   3.696  90.000 ns
add_accum_w24_n_stages6                177545   3.061  90.000 ns

add_pipe_w24_n_stages1                  84351   1.114   0.100 ns
add_pipe_w24_n_stages2                  83959   1.249   0.100 ns
add_pipe_w24_n_stages3                 103383   1.105   0.100 ns
add_pipe_w24_n_stages4                 117358   1.001   0.100 ns
add_pipe_w24_n_stages6                 150854   0.896   0.100 ns

add_accum_w24_n_stages1                150738   2.023   0.100 ns
add_accum_w24_n_stages2                149544   1.757   0.100 ns
add_accum_w24_n_stages3                183994   1.514   0.100 ns
add_accum_w24_n_stages4                191611   1.444   0.100 ns
add_accum_w24_n_stages6                224175   1.332   0.100 ns
Normal exit.
`endif


module add_pipe
  #( int w = 21, n_stages = 3 )
   ( output uwire [w-1:0] sum,
     input uwire [w-1:0] a, b,
     input uwire clk );

   localparam int bits_per_stage = ( w + n_stages - 1 ) / n_stages;
   localparam int wr = n_stages * bits_per_stage; // w rounded.

   logic [wr-1:0] pl_a[n_stages+1], pl_b[n_stages+1], pl_sum[n_stages+1];
   logic pl_carry[n_stages+1];
```

```
   always_ff @( posedge clk ) begin

      pl_a[0] = a;
      pl_b[0] = b;
      pl_carry[0] = 0;

      for ( int s=0; s<n_stages; s++ ) begin

         automatic logic [bits_per_stage:0] sumi =
           pl_a[s][bits_per_stage-1:0] +
             pl_b[s][bits_per_stage-1:0] + pl_carry[s];

         pl_carry[s+1] <= sumi[bits_per_stage];
         pl_sum[s+1] <=
           { sumi[bits_per_stage-1:0], pl_sum[s] } >> bits_per_stage;
         pl_a[s+1] <= pl_a[s] >> bits_per_stage;
         pl_b[s+1] <= pl_b[s] >> bits_per_stage;

      end

   end

   assign sum = pl_sum[ n_stages ][w-1:0];

endmodule



// cadence translate_off


program reactivate
   (output uwire clk_reactive, output int cycle_reactive,
    input uwire clk, input var int cycle);
   assign clk_reactive = clk;
   assign cycle_reactive = cycle;
endprogram

module testbench;

   localparam int n_stages[] = { 1, 2, 3, 5, 6 };

   localparam int nw = 5; // Cadence, please fix this.
   initial if ( nw != n_stages.size() )
     $fatal(1,"Constant nw should be %0d.\n",n_stages.size() );

   int t_errs;     // Total number of errors.
   initial t_errs = 0;
   final $write("Total number of errors: %0d\n",t_errs);

   uwire d[nw:-1];    // Start / Done signals.
   assign d[-1] = 1;  // Initialize first at true.

   // Instantiate a testbench at each size.
   //
   for ( genvar i=0; i<nw; i++ )
     testbench_n #(n_stages[i]) t2( .done(d[i]), .tstart(d[i-1]) );

endmodule


module testbench_n
  #( int n_stages = 3 )
   ( output logic done, input uwire tstart );

   localparam int n_tests = 10000;
   localparam int w = 30;

   localparam int a_in_max = 42;
   localparam int cyc_max = 1 << 30;

   localparam int lat_limit_empty = n_stages + 2;
```

```
   localparam int lat_min_empty = n_stages;
   localparam int lat_limit_full = 2 + (1+$clog2(n_stages)) * ( n_stages + 1 );

   bit clk;
   int cycle, cycle_limit;
   logic clk_reactive;
   int cycle_reactive;
   reactivate ra(clk_reactive,cycle_reactive,clk,cycle);

   string event_trace;

   initial begin
      clk = 0;
      cycle = 0;

      done = 0;
      cycle_limit = cyc_max;
      wait( tstart );

      fork
         while ( !done ) #1 cycle += clk++;
         wait( cycle >= cycle_limit )
           $write("Exit from clock loop at cycle %0d, limit %0d.  %s\n %s\n",
                  cycle, cycle_limit, "** CYCLE LIMIT EXCEEDED **",
                  event_trace);
      join_any;

      done = 1;
   end

   uwire [w-1:0] sum;
   uwire sum_valid;
   logic [w-1:0] a;
   logic a_valid, reset;

   add_accum #(w,n_stages) fpa(sum, sum_valid, a, a_valid, reset, clk);

   int rsum;
   bit tests_start;
   int series_idx, value_idx, series_n_vals;
   int n_errs, n_underdue_errs, n_overdue_errs, n_tests_done;
   int sum_due_cyc_earliest, sum_due_cyc, n_correct;
   int last_a_cyc;
   int latency_sum, latency_sum_n;
   bit error_val_issued, error_late_issued;

   initial wait ( done ) begin
      automatic int not_done = n_tests - series_idx;
      $write("Done with %0d-stage tests, %0d series.\n Correct, %0d.   Errors: %0d not done, %0d val, %0d/%0d early/late.\n",
             n_stages, series_idx,
             n_correct, not_done, n_errs, n_underdue_errs, n_overdue_errs );
      $write("For %0d stages average latency %.2f cycles.\n",
             n_stages,
             real'(latency_sum) / ( latency_sum_n ? latency_sum_n : 1 ) );
      testbench.t_errs += n_errs + n_underdue_errs + n_overdue_errs + not_done;
   end

   initial begin

      wait( tests_start );

      while ( !done ) @( posedge clk_reactive ) begin

         if ( sum_valid ) begin

            automatic bit pending = sum_due_cyc < cyc_max;

            if ( pending ) begin
               n_tests_done++;
               sum_due_cyc = cyc_max;
               if ( sum === rsum ) n_correct++;
               latency_sum += cycle - last_a_cyc;
               latency_sum_n++;
               if ( cycle < sum_due_cyc_earliest ) begin
```

```
                n_underdue_errs++;
                if ( n_underdue_errs < 5 ) begin
                   $write
                     ("At cyc %0d, value ready too soon, %0d, cyc. (Min cyc %0d.)\n",
                      cycle, last_a_cyc - cycle, lat_limit_empty
                      );
                   if ( event_trace != "" ) $write(" %s\n",event_trace);
                end

             end
          end

          if ( !error_val_issued && sum !== rsum ) begin
             error_val_issued = 1;
             n_errs++;
             if ( n_errs < 5 ) begin
                $write("At cyc %0d, wrong sum, %0d != %g (correct)\n",
                       cycle, sum, rsum);
                if ( event_trace != "" ) $write(" %s\n",event_trace);
             end
          end

       end else if ( sum_due_cyc <= cycle ) begin

          if ( !error_late_issued ) begin
             error_late_issued = 1;
             n_overdue_errs++;
             sum_due_cyc = cyc_max;
             if ( n_overdue_errs < 5 ) begin
                $write("At cycle %0d, sum overdue.\n",cycle);
                if ( event_trace != "" ) $write(" %s\n",event_trace);
             end
          end
       end

    end

end

initial begin

    automatic int seed = 4755;
    automatic int series_sparsity = 0;
    rsum = 0;
    n_errs = 0;
    latency_sum_n = 0;
    latency_sum = 0;
    error_val_issued = 0;
    error_late_issued = 1;
    series_idx = 0;
    value_idx = 0;
    series_n_vals = 0;
    n_overdue_errs = 0;
    n_underdue_errs = 0;
    sum_due_cyc = cyc_max;
    sum_due_cyc_earliest = 0;
    n_tests_done = 0;
    n_correct = 0;
    event_trace = "";

    wait( tstart );
    $write("Starting tests for %0d-stage pipeline.\n",n_stages);

    @( negedge clk );
    reset = 1;
    event_trace = $sformatf("R(%0d)",cycle);
    a_valid = 0;
    a = 0;
    @( negedge clk );
    cycle_limit = cycle + n_stages * 2;
    tests_start = 1;
    reset = 0;
    @( negedge clk );
```

```
      wait( sum_valid );

      while ( series_idx < n_tests ) begin

         @( negedge clk );

         a = $dist_uniform( seed, 1, a_in_max );

         if ( value_idx >= series_n_vals ) begin

            a_valid = 0;

            if ( sum_valid ) begin

               series_idx++;
               value_idx = 0;
               event_trace = $sformatf("R(%0d)",cycle);
               reset = 1;
               a_valid = 0;
               rsum = 0;
               series_n_vals = $dist_uniform( seed, 1, 10 );
               series_sparsity = series_idx % 6;
               sum_due_cyc = cycle + 1;
               sum_due_cyc_earliest = cycle;
               error_val_issued = 0;
               error_late_issued = 0;
               cycle_limit = cycle + 1;
            end

         end else begin

            reset = 0;
            a_valid = series_sparsity == 0
              || $dist_uniform( seed, 0, series_sparsity ) == 0;
            cycle_limit = cycle + lat_limit_full;

         end

         if ( a_valid ) begin
            value_idx++;
            event_trace = {event_trace,$sformatf("+%0d(%0d)",a,cycle)};
            error_val_issued = 0;
            error_late_issued = 0;
            rsum += a;
            last_a_cyc = cycle;
            sum_due_cyc = cycle +
              ( sum_valid ? lat_limit_empty : lat_limit_full );
            sum_due_cyc_earliest =
              cycle + ( value_idx > 1 ? lat_min_empty : 0 );
         end

      end

      done = 1;

   end

endmodule

// cadence translate_on
```

20 Fall 2018 Solutions

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2018 Homework 1 -- SOLUTION
//

 /// Assignment https://www.ece.lsu.edu/koppel/v/2018/hw01.pdf

`default_nettype none

////////////////////////////////////////////////////////////////////////////////
///  Problem 1
//
 /// Modify sort2 so that it implements a 2-input sorting network ..
 /// .. using explicity structural code only.
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Use structural code only: including mux2 and compare_le.
//     [✔] Pay attention to bit widths.
//     [✔] Module must be synthesizable.

module sort2
  #( int w = 8 )
   ( output uwire [w-1:0] x0, x1,
     input uwire [w-1:0] a0, a1 );

   /// SOLUTION
   //
   //  The only common problem was forgetting to specify the width.

   uwire c;
   compare_le #(w) comp(c, a0, a1);

   mux2 #(w) m0(x1,c,a0,a1);
   mux2 #(w) m1(x0,c,a1,a0);

endmodule

module mux2
  #( int w = 4 )
   ( output uwire [w-1:0] x,
     input uwire select,
     input uwire [w-1:0] a0, a1 );
   assign x = select ? a1 : a0;
endmodule

module compare_le
  #( int w = 2 )
   ( output uwire c,
     input uwire [w-1:0] a, b );
   assign c = a <= b;
endmodule


module sort2_is
  #( int w = 8 )
   ( output uwire [w-1:0] x0, x1,
     input uwire [w-1:0] a0, a1 );

   assign {x0, x1} = a0 <= a1 ? { a0, a1 } : { a1, a0 };

endmodule
```

```
////////////////////////////////////////////////////////////////////////////////
/// Problem 2
//
 /// Modify sort4 so that it implements a 4-input sorting network ..
 /// .. using explicity structural code only.
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Use structural code only: including sort2 and sort3 modules.
//     [✔] Pay attention to bit widths.
//     [✔] Module must be synthesizable.
//     [✔] The critical path length should be 4 or fewer sort2 modules.


module sort4
  #( int w = 8 )
   ( output uwire [w-1:0] x0, x1, x2, x3,
     input uwire [w-1:0] a0, a1, a2, a3 );

   /// SOLUTION

   uwire [w-1:0] s10, s11, s12, s13, s20;

   // Connect sort2 modules into a tree, so that the longest path
   // is through 2 sort2 modules.
   //
   sort2 #(w) s2( s10, s11,  a0, a1 );
   sort2 #(w) s3( s12, s13,  a2, a3 );
   sort2 #(w) s4( s20, x3,  s11, s13 );

   // Note that s20 will be available later than s10 and s12 because
   // the path from a0 - a3 to s20 go through two sort2 modules,
   // whereas the paths from a0 - a3 to s10 and s12 only pass through
   // one sort2 module. Since s20 will be available later connect it
   // to sort3 port a2, which is closer to the outputs.
   //
   sort3 #(w) s1( x0, x1, x2,  s10, s12, s20 );
   //
   // Note that module would be slower if s12 and s20 connections above
   // were reversed:
   //  sort3 #(w) s1( x0, x1, x2,  s10, s20, s12 );

endmodule


module sort3
  #( int w = 8 )
   ( output uwire [w-1:0] x0, x1, x2,
     input uwire [w-1:0] a0, a1, a2 );

   uwire [w-1:0] i10, i11, i21;

   sort2 #(w) s0_01( i10, i11,   a0, a1 );
   sort2 #(w) s1_12( i21, x2,  i11, a2  );
   sort2 #(w) s2_01( x0, x1,  i10, i21  );

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Testbench Code
//
//  The code below instantiates some of the modules above,
//  provides test inputs, and verifies the outputs.
```

```
//
//  The testbench may be modified to facilitate your solution. For
//  example, one might modify the testbench so that the first tests it
//  performs are those which make it easier to determine what the
//  problem is, for example, test inputs that are all 0's or all 1's.
//
//  Of course, the removal of tests which your module fails is not a
//  method of fixing a broken module.  The TA-bot will test your
//  code using a fresh copy of the testbench, not the one below.


// cadence translate_off

module sortx
  #( int max_size = 5,
     int modnum = 0,
     int w = 8,
     int max_muts = 3)
   ( output uwire [w-1:0] xlong[max_muts][max_size],
     input uwire [w-1:0] a[max_size] );

   uwire [w-1:0] x[max_size];
   assign xlong[modnum] = x;

   if ( modnum == 0 ) begin:A

      localparam int s_size = 2;
      localparam string name = "sort2";
      sort2 #(w) s(x[0],x[1],a[0],a[1]);

   end else if ( modnum == 1 ) begin:A

      localparam int s_size = 2;
      localparam string name = "sort2_is";
      sort2_is #(w) s(x[0],x[1],a[0],a[1]);

   end else if ( modnum == 2 ) begin:A

      localparam int s_size = 3;
      localparam string name = "sort3";
      sort3 #(w) s(x[0],x[1],x[2],a[0],a[1],a[2]);

   end else begin:A

      localparam int s_size = 4;
      localparam string name = "sort4";
      sort4 #(w) s(x[0],x[1],x[2],x[3],a[0],a[1],a[2],a[3]);

   end

endmodule


module testbench;

   localparam int w = 8;
   localparam int n_tests = 100;
   localparam int max_size = 4;
   localparam int max_muts = 4;

   logic [w-1:0] a[max_size];
   uwire [w-1:0] x[max_muts][max_size];

   typedef struct { int idx; string name; int s_size; } Info;
   Info pi[$];
```

```
   for ( genvar i=0; i<4; i++ ) begin
      sortx #(max_size,i,w,max_muts) s(x,a);
      initial pi.push_back( '{ i, s.A.name, s.A.s_size} );
   end

   initial begin

      automatic int g_elt_err_count = 0;
      automatic int g_sort_err_count = 0;

      $write("Starting testbench.");

      // Initialize the input to a recognizable pattern, which should
      // be overwritten but if not, we can tell. If we print the value in
      // hex.
      for ( int e = 0; e < max_size; e++ ) a[e] = 'haaaaaaaa;

      foreach ( pi[idx] ) begin

         automatic Info p = pi[idx];
         automatic string mut = p.name;
         automatic int s_size = p.s_size;
         automatic logic [w-1:0] shadow[] = new[s_size];

         for ( int i = 0;  i < n_tests;  i++ ) begin

            automatic int this_elt_err_count = 0;

            // To make sure that the comparison is correct restrict the
            // key to a subset of bits.
            automatic int n_bits = {$random} % w + 1;
            automatic int mask = ( 1 << n_bits ) - 1;

            for ( int i=0; i<w; i++ ) begin
               automatic int b = {$random} % w;
               {mask[b],mask[i]} = {mask[i],mask[b]};
            end

            for ( int e = 0; e < s_size; e++ )
              begin a[e] = {$random} & mask; shadow[e] = a[e]; end

            #1;

            shadow.sort();

            for ( int e = 0; e < s_size; e++ ) begin
               automatic logic [w-1:0] elt = x[p.idx][e];
               if ( shadow[e] === elt ) continue;
               this_elt_err_count++;
               g_elt_err_count++;
               if ( g_elt_err_count > 5 ) continue;
               $write
                 ("Mod %s, sort %2d index %2d, wrong elt %d != %d (correct)\n",
                  mut,i, e, elt, shadow[e]);
            end

            if ( this_elt_err_count ) g_sort_err_count++;

         end

         $write("Tests for %s done, errors in %d of %d sorts.\n",
                mut,  g_sort_err_count, n_tests);

      end
```

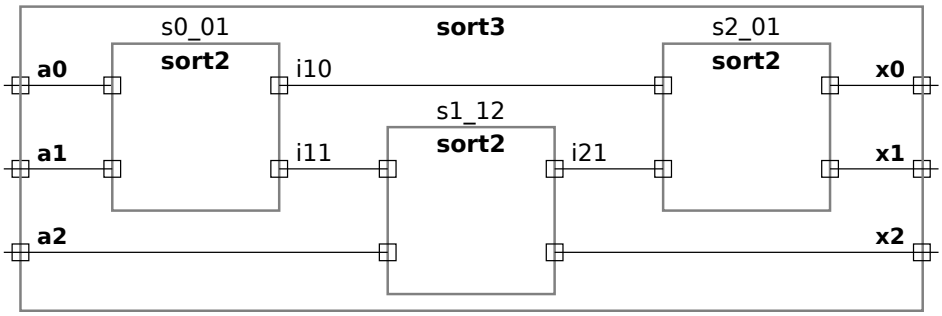
```
    end
endmodule
// cadence translate_on
```

LSU EE 4755 Homework 2 Solution Due: 12 September 2018

Problem 1: The Verilog code below is the `sort3` module from Homework 1. Draw a diagram of the hardware as described by `sort3`, showing the `sort2` modules as boxes. Be sure to label the input and output ports with the same symbols used in the module.

```
module sort3
  #( int w = 8 )
   ( output uwire [w-1:0] x0, x1, x2,
     input uwire [w-1:0] a0, a1, a2 );

   uwire [w-1:0] i10, i11, i21;

   sort2 #(w) s0_01( i10, i11,   a0, a1 );
   sort2 #(w) s1_12( i21, x2,  i11, a2  );
   sort2 #(w) s2_01( x0, x1,  i10, i21  );

endmodule
```

Solution appears below.



Problem 2: It is possible to build an n-element sorting network using $\frac{n}{2}\lg^2 n$ two-element sorting networks in such a way that the n-element sorting network has a critical path of $\lg^2 n$. (Note: $\lg n \equiv \log_2 n$.) But this assignment is concerned with n-element sorting networks using $n(n-1)/2$ two-element sorting networks, which we will call n-element *bad sorting networks* or *bad sorters* for short.

An n-element bad sorter has inputs $a_0, a_1, \ldots, a_{n-1}$ and outputs $x_0, x_1, \ldots, x_{n-1}$. The largest value is routed to x_{n-1}.

A 2-element bad sorter is a single `sort2` module. An n-element bad sorter, $n > 2$, can be constructed using an $(n-1)$-element bad sorter and $n-1$ `sort2` modules as follows. The $n-1$ `sort2` modules are connected to the n inputs and to each other in such a way that the largest value is routed to a specific output of one of the `sort2` modules. That specific `sort2` output is connected to output x_{n-1} of the n-element sorter. The other values connect to the $(n-1)$-element bad sorter, and the $(n-1)$-element bad sorter outputs connect to outputs $x_0, x_1, \ldots, x_{n-2}$ of the n-element bad sorter that we are constructing. Note that this generalizes the solution to Homework 1 Problem 2.

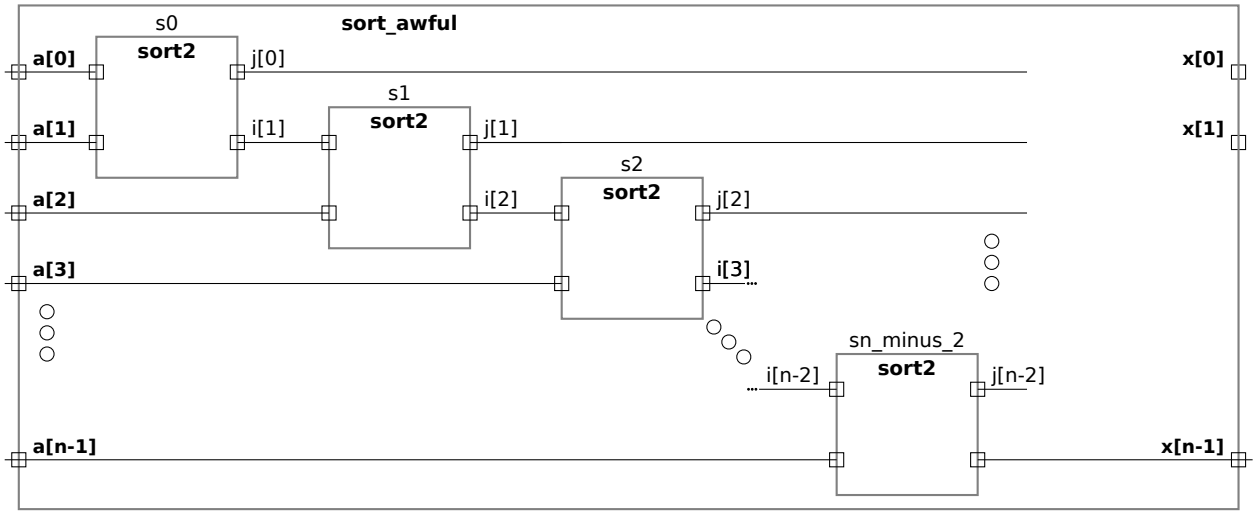
The description above is recursive. At level i (the same as n above) another $i-1$ `sort2` modules are used. For a 4-element sorter we need $(4-1)+(3-1)+1 = 6$ `sort2` modules. The

cost of an n-element bad sorter is found by solving the summation $\sum_{i=2}^{n} i-1$, which is $n(n-1)/2$. That's $O(n^2)$, which is how the bad sorter got its name.

It gets worse. The critical path through the bad sorter can range from bad to awful. That depends on two things: How the `sort2` modules are used to find the largest value, and how the `sort2` modules connect to the $(n-1)$-element bad sorter.

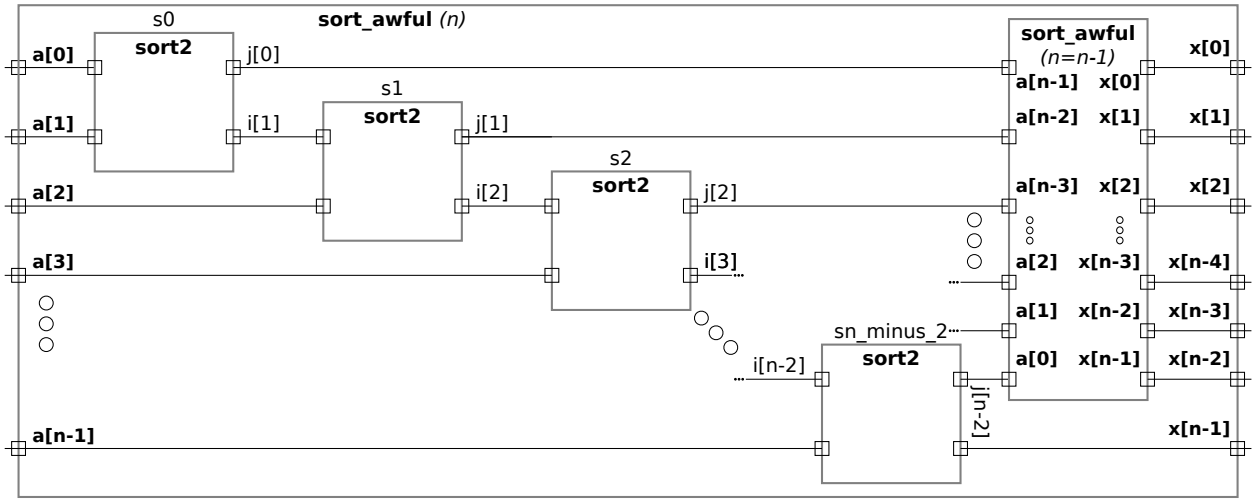
(a) Show the worst way that `sort2` modules can be connected to find the largest value. *Hint: the critical path should be* $n-1$ `sort2` *modules.* Provide a sketch for the general case, and an example for $n=4$.

Call the `sort2` modules s_0 to s_{n-2}. Connecting output `x1` of s_i to input `a0` of s_{i+1} for $0 \le i < n-1$ is the worst way to connect $n-1$ modules. See the illustration below. The critical path starts at `a0` or `a1` and ends at either output of sorter s_{n-2}.



(b) Show the worst way that the `sort2` modules, as connected above, can connect to the $(n-1)$-element sorter. Provide a sketch.

Notice that the critical path through the `sort2` modules starts at `a[0]` and ends at `j[n-2]`. So when connecting `j[n-2]` to the smaller sort module the worst ports that it can be connected to are `a[0]` and `a[1]`. That's shown below. Note that `n` inside the smaller `sort_awful` is equal to `n-1` in the larger one.

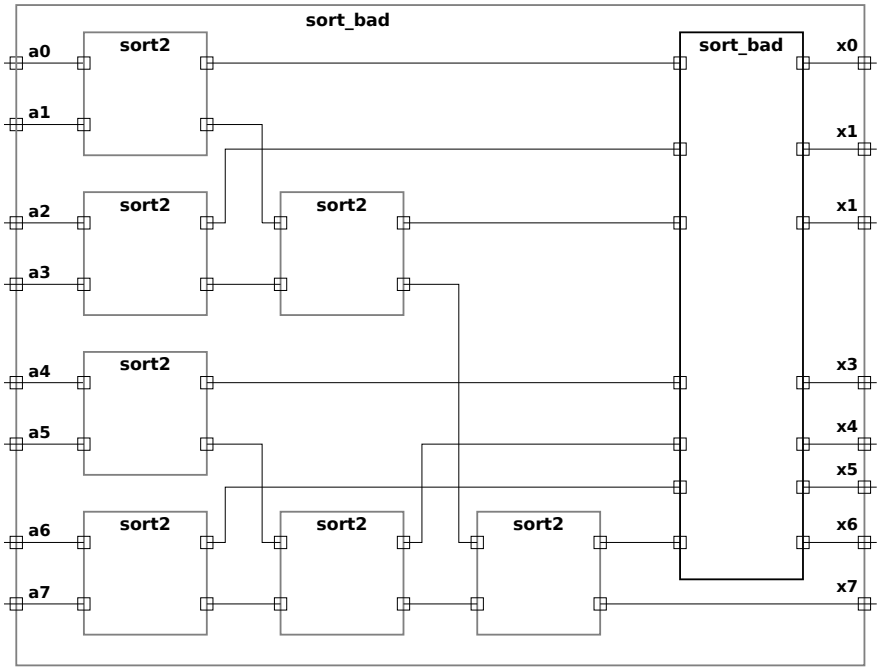


(*c*) Determine the critical path for an n-element bad sorter constructed in the way described in the last two parts. *Hint: The math part should be familiar.*

In an n-input awful bad sorter the path from `a0` to `j[n-2]` is of length $n-1$. Signal `j[n-2]` connects to `a0` of an $n-1$ element bad sorter where it goes through $n-2$ `sort2` modules. The total length of the path is $\sum_{i=n}^{2} i - 1 = n(n-2)/2$ `sort2` modules.

(*d*) Show a much better way of connecting the `sort2` modules to find the largest value. It should be easy to show that the critical path is the lowest that is possible. Provide a sketch for $n = 8$.

Connect the `sort2` modules in a tree, the solution appears below for $n = 8$, and showing the recursive connection.



The problem with the approach to building the bad sorters described in this assignment is that each level in the recursion reduces the size by 1 (that is, from n to $n-1$), and so the critical path must be at least $O(n)$. As some students may have realized, a better approach would be to use recursion in which the n inputs were split between two $\frac{n}{2}$-element networks and then somehow combined. But how? The key insight, described by K. E. Batcher in a landmark 1968 paper, is not to try to recursively describe a sorting network, but to instead recursively describe a network that merges two already sorted sequences. The input to a 2-element merge network would be two 1-element sorted sequences. (Of course, every 1-element sequence is sorted.) Pairs of 2-element merge networks feed a 4-element merge network, and so on. This will be further described later in the semester.

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2018 Homework 3 -- SOLUTION
//

 /// Assignment https://www.ece.lsu.edu/koppel/v/2018/hw03.pdf

`default_nettype none

////////////////////////////////////////////////////////////////////////////////
/// All Problems
//
 /// Modify sort2 so that it implements a 2-input sorting network ..
 /// .. using explicit and implicit structural code only.
//
//     [✔] Easy:   Modify to compare keys, not data.
//     [✔] Easy:   Modify to sort pairs of signed integer keys.
//     [✔] Medium: Modify to sort pairs of floating-point keys.
//     [✔] Medium: Modify to sort one signed int and one FP key.
//     [✔] Hard:   Keep cost and critical path low.
//
//     [✔] Use implicit and explicit structural code only.
//     [✔] Use mux2 to swap the values.
//     [✔] Add ChipWare Verilog modules to includes at the end of this file.
//
//     [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//     [✔] Make sure that the testbench does not report errors.
//
//     [✔] Use SimVision for debugging.
//         DMK: Yes, I used SimVision.
//     [✔] Modify testbench to facilitate solution ..
//         .. but code must pass original testbench.

module sort2
  #( int w = 30,
     int k = 16,
     int exp = 5,
     int sig = k - exp - 1 )
   ( output uwire [w-1:0] x0, x1,
     input uwire [w-1:0] a0, a1 );

   /// Encoding of a0 and a1
   //
   // Bits        Contents
   // ---------   ----------------------------------------
   // w-1 : k+1   Data
   // k           Key type: 0, integer;  1, floating point.
   // k-1 : 0     Key, a signed integer; if Key type = 0.
   // k-1         Sign bit, if key type = 1.
   // k-2 : sig   Exponent, if key type = 1.
   // sig-1 : 0   Significand, if key type = 1.

   /// SOLUTION OVERVIEW
   //
   //  Compare integer/integer keys using <= operator.
   //  Compare fp/fp and fp/integer keys using ChipWare comparison operator.

   //  For fp/integer keys that are reported equal use the inexact
   //  status bit.

   // Extract key portion of inputs and assign to a logic signed type.
   //
   uwire signed [k-1:0] k0 = a0[k-1:0];
   uwire signed [k-1:0] k1 = a1[k-1:0];

   // Integer / Integer Comparison
   //
   uwire ci = k0 <= k1;
```

```
//
// Note that ci is ignored if either input is FP.


//
/// Convert Keys to FP
//

// Floating-point version of keys. Only valid if key is an integer.
//
uwire [k-1:0] kif0, kif1; // Key Integer converted to Float.

uwire [7:0] si0, si1;  // Status output of integer-to-FP modules.
localparam logic [2:0] rnd_to_0 = 3'b1;

// Convert using ChipWare integer two floating-point modules.
//
CW_fp_i2flt #( .sig_width(sig), .exp_width(exp), .isize(k), .isign(1) )
  itof0( .z(kif0), .status(si0), .a(k0), .rnd(rnd_to_0) );
CW_fp_i2flt #( .sig_width(sig), .exp_width(exp), .isize(k), .isign(1) )
  itof1( .z(kif1), .status(si1), .a(k1), .rnd(rnd_to_0) );

// Extract the inexact bit.
//
uwire inexact0 = si0[5], inexact1 = si1[5];
//
// If this bit is 1 the converted value is slightly less in
// magnitude than the original integer value. Less because the
// round-towards-zero (rnd_to_0) rounding option was selected.

// Select the FP version of the key.
//
uwire [k-1:0] fk0 = a0[k] ? k0 : kif0;
uwire [k-1:0] fk1 = a1[k] ? k1 : kif1;
//
// Note that k0 is selected if the key in a is already FP,
// otherwise use the output of the conversion module.


//
/// Floating Point Comparison
//
uwire gt, lt, eq, un;
uwire [k-1:0] z0, z1; // Unused
uwire [7:0] s0, s1;

CW_fp_cmp #( .sig_width(sig), .exp_width(exp), .ieee_compliance(0) )
fp_comp(.a(fk0),.b(fk1),.agtb(gt),.altb(lt),.aeqb(eq),
        .zctr(1'b0), .unordered(un),.z0(z0),.z1(z1),
        .status0(s0), .status1(s1));

/// Check whether kif0 <= kif1.
//
//  Approximate method:
//
uwire cf_approx = lt || eq;  // Unused, shown to explain solution.
//
// cf_approx can be wrong when eq is true because of rounding
// errors when converting an integer to FP. Wire inexact0 is 1 when
// there was a rounding error converting k0, etc. If eq is 1 then
// a0 <= a1 iff one of the two cases below is true:
//
// Case 1:
//   k0 is negative
//     k0 is FP (so no rounding)
//     or k0 is an integer and no rounding error in the conversion.
//     -- otherwise kif0 is larger than the value in a0
//
// Case 2:
//   k0 is positive
```

```
   //     k1 is FP (so no rounding)
   //     or k1 is an integer and no rounding error in the conversion.
   //     -- otherwise kif1 is smaller than the value in a1.

   // Determine whether a0 <= a1 accounting for rounding errors, as
   // described above.
   //
   uwire cf = lt || eq &&
         (   !kif0[k-1] && ( a0[k] || !inexact0 ) // Case 1
           || kif0[k-1] && ( a1[k] || !inexact1 ) // Case 2
             );

   // If at least one input is FP use FP comparison result, else int result.
   //
   uwire c = a0[k] || a1[k] ? cf : ci;

   mux2 #(w) m0(x0,c,a1,a0);
   mux2 #(w) m1(x1,c,a0,a1);

endmodule

module mux2
  #( int w = 4 )
   ( output uwire [w-1:0] x,
     input uwire select,
     input uwire [w-1:0] a0, a1 );
   assign x = select ? a1 : a0;
endmodule



////////////////////////////////////////////////////////////////////////////////
/// Testbench Code
//
//  The code below instantiates some of the modules above,
//  provides test inputs, and verifies the outputs.
//
//  The testbench may be modified to facilitate your solution. For
//  example, one might modify the testbench so that the first tests it
//  performs are those which make it easier to determine what the
//  problem is, for example, test inputs that are all 0's or all 1's.
//
//  Of course, the removal of tests which your module fails is not a
//  method of fixing a broken module.  The TA-bot will test your
//  code using a fresh copy of the testbench, not the one below.


// cadence translate_off

module testbench;

   var bit s[3];
   testbench_size #(32,16,6) t1(s[1],s[0]);
   testbench_size #(24,14,5) t2(s[2],s[1]);

   initial begin
      s[0] = 1;
      wait( s[2] );
      $write("\nAll done.\n");
   end

endmodule



module testbench_size
  #( int w = 30,
     int k = 16,
```

```
  int exp = 6,
  int sig_width = k - exp - 1 )
( output var bit done, input var bit start );

localparam int s_pos = k - 1;
localparam int exp_hi = s_pos - 1;
localparam int exp_lo = s_pos - exp;
localparam int sig_hi = exp_lo - 1;
localparam int bias = ( 1 << exp - 1 ) - 1;
localparam int exp_i_max = bias + k - 1;
localparam int exp_max = ( 1 << exp ) - 2;
localparam int exp_range_ini = exp_i_max - bias;
localparam int exp_range_gti = exp_max - exp_i_max - 1;

function real fp_to_val(input logic [k-1:0] a);
   fp_to_val =
     a[exp_hi:0] == 0 ? 0.0 :
       ( ( 1.0 + a[sig_hi:0] / real'( 1 << sig_width ) )
         * 2 ** ( 0.0 + a[exp_hi:exp_lo]-bias )
         * ( a[s_pos] ? -1 : 1 ) );
endfunction

localparam int num_tests = 3000000;
localparam int test_ff_start = num_tests / 3;
localparam int test_if_start = test_ff_start * 2;

uwire [w-1:0] x0, x1;
logic [w-1:0] a[2];
sort2 #(w,k,exp,sig_width) s2(x0,x1, a[0], a[1]);

initial begin

   automatic int err_count[string] = '{"ii":0, "ff":0, "if":0 };

   automatic logic [1:0][w-1:0] tests[$];
   /// Add tests below by copying output of testbench.
   //  Note: Tests only work at a particular value of exp, k.
   //  Put in correct place.
   case ( k )
     14: begin
        tests.push_back('h2a823); tests.push_back('h77b7e);
     end
     16: begin
        tests.push_back('hdc641209); tests.push_back('ha0935641);
     end
   endcase

   wait( start );
   $write("Starting testbench for w=%0d, k=%0d, exp=%0d  sig width=%0d...\n",
          w, k, exp, exp_lo);

   for ( int i=0; i<num_tests; i++ ) begin

      automatic logic [k-1:0] i_1_mask = 1 << {$random} % k;
      automatic string test_type =
        i < test_ff_start ? "ii" :
          i < test_if_start ? "ff" : "if";

      bit fp[2];
      real val[2];
      bit swap;
      logic [w-1:0] shadow_x0, shadow_x1;

      for ( int j=0; j<2; j++ )
        fp[j] = test_type == "ff" || test_type == "if" && ( (i+j) & 1 );

      // FP Options: 0, (0,1), int range, > int range, int, int, int, int

      for ( int j=0; j<2; j++ ) begin
```

```
   automatic int fp_sz = fp[j] ? {$random} % 4 : 4;
   a[j][w-1:0] = {$random};
   a[j][k] = fp[j];

   case ( fp_sz )
     0: a[j][exp_hi:0] = 0;
     1: a[j][exp_hi:exp_lo] = 1 + {$random} % bias;
     2: a[j][exp_hi:exp_lo] = bias + {$random} % exp_range_ini;
     3: a[j][exp_hi:exp_lo] = exp_i_max + {$random} % exp_range_gti;
     default:; // For int leave random value.
   endcase
end

// Test cases for floating-point pairs.
if ( a[0][k] && a[1][k] && a[1][exp_hi:0] ) begin
   // Generate fp numbers with matching exponents
   if ( {$random} & 1 ) a[0][exp_hi:exp_lo] = a[1][exp_hi:exp_lo];
   // Generate fp numbers with matching significands.
   if ( {$random} & 1 ) a[0][sig_hi:0] = a[1][sig_hi:0];
end

// Test cases for integer pairs.
if ( !a[0][k] && !a[1][k] ) begin
   case ( {$random} % 6 )

     // Differ by 1
     0: a[1][k-1:0] = a[0][k-1:0] - 1;
     1: a[1][k-1:0] = a[0][k-1:0] + 1;

     // Differ by 2
     2: a[1][k-1:0] = a[0][k-1:0] - 2;
     3: a[1][k-1:0] = a[0][k-1:0] + 2;

     // Sort key in only 1 bit.
     4: begin a[0][k-1:0] &= i_1_mask; a[1][k-1:0] &= i_1_mask; end

     default:;
   endcase
end

// Test cases for int/fp keys.
if ( a[0][k] != a[1][k] ) begin
   automatic int opt = {$random} % 32;
   casex ( opt )
     'h0xxx: if ( a[0][k] ) a[1][k-1:0] = fp_to_val(a[0])+opt[2:0]-4;
     'h1xxx: if ( a[1][k] ) a[0][k-1:0] = fp_to_val(a[1])+opt[2:0]-4;
   endcase

end

// Replace the keys found above with user-defined keys, if any.
if ( tests.size() ) begin
   a[0] = tests.pop_front();
   a[1] = tests.pop_front();
end

case ( {a[0][k], a[1][k] } )
  'b00: test_type = "ii";
  'b11: test_type = "ff";
  default: test_type = "if";
endcase

for ( int j=0; j<2; j++ )
  val[j] = a[j][k] ? fp_to_val(a[j]) : signed'(a[j][s_pos:0]);

swap = val[0] > val[1];
{ shadow_x0, shadow_x1 } = { a[swap], a[1-swap] };

#1;
```

```
          if ( shadow_x0 !== x0 || shadow_x1 !== x1 ) begin

             err_count[test_type]++;
             if ( err_count[test_type] > 5 ) continue;

             $write
               ("Test %s %4d, error (x0,x1): (%h,%h) != (%h,%h) correct.\n",
                test_type, i,
                x0, x1, shadow_x0, shadow_x1);
             for ( int j=0; j<2; j++ )
               $write("           a%1d: data %h, key %12.5f = %s %s\n",
                      j, a[j][w-1:k], val[j], a[j][k] ? "FP " : "INT",
                      a[j][k] ?
                      $sformatf("s %b  exp %0d-%0d=%0d  sig 'h%h",
                                a[j][s_pos],
                                a[j][exp_hi:exp_lo], bias,
                                signed'({1'b0,a[j][exp_hi:exp_lo]}) - bias,
                                a[j][sig_hi:0])
                      : $sformatf("'h%h",a[j][k-1:0]));
             $write("           To re-run paste:  tests.push_back('h%h); tests.push_back('h%h);\n",
                    a[0],a[1]);
          end

       end

       $write("Done with %0d tests for k=%0d, exp=%0d:", num_tests,k,exp);
       foreach ( err_count[et] )
         $write("   %0d %s errs,", err_count[et], et);
       $write("\n");

       done = 1;

    end

endmodule



// cadence translate_on

`default_nettype wire

`include "/apps/linux/cadence/GENUS171/share/synth/lib/chipware/sim/verilog/CW/CW_fp_cmp.v"
`include "/apps/linux/cadence/GENUS171/share/synth/lib/chipware/sim/verilog/CW/CW_fp_i2flt.v"
```

LSU EE 4755 Homework 4 Solution Due: 3 October 2018

Problem 1: Solve 2017 Final Exam Problem 3, in which the cost and delay of two alternative designs are to be compared.

See posted final exam solution on the previous work page at `https://www.ece.lsu.edu/koppel/v/prev.html`.

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2018 Homework 5 -- SOLUTION
//

 /// Assignment https://www.ece.lsu.edu/koppel/v/2018/hw05.pdf


`default_nettype none
////////////////////////////////////////////////////////////////////////////////
///  Problem 1
//
 /// Complete batcher_sort so that it recursively implements a Batcher
 /// sorter using a merge module.
//
//     [✔] Assume that n is a power of 2.
//     [✔] Use implicit and explicit structural code only.
//     [✔] Use recursion as described in the handout.
//     [✔] Use behav_merge initially and when it's done, batcher_merge.
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//
//     [✔] Use SimVision for debugging.
//     [✔] Modify testbench to facilitate solution ..
//         .. but code must pass original testbench.
//
//     [✔] As always, code should be efficient and clearly written.

module batcher_sort
  #( int n = 4, int w = 8 )
   ( output uwire [w-1:0] x[n], input uwire [w-1:0] a[n] );

   /// SOLUTION

   if ( n == 1 ) begin

      // Set the terminal case at n==1 ..
      // .. because sorting is easy when there's just one element!
      //
      assign x = a;

   end else begin

      localparam int nh = n/2;
      uwire [w-1:0] xlo[nh], xhi[nh];

      // Recursively instantiate two sorters, slo and shi, ..
      // .. slo will sort elements 0 to nh-1, and ..
      // .. shi will sort elements nh to n-1.
      //
      batcher_sort #(nh,w) slo( xlo, a[0:nh-1] );
      batcher_sort #(nh,w) shi( xhi, a[nh:n-1] );

      // Use a merge module to combine the two sorted sequences.
      //
      batcher_merge #(nh,w) m( x, xlo, xhi );

   end

endmodule
```

```
module behav_merge
  #( int n = 4, int w = 8 )
   ( output logic [w-1:0] x[2*n], input uwire [w-1:0] a[n], b[n] );

   logic [$clog2(n+1)-1:0] ia, ib;
   always_comb begin
      ia = 0; ib = 0;
      for ( int i = 0;  i < 2*n;  i++ )
        x[i] = ib == n || ia < n && a[ia] <= b[ib] ? a[ia++] : b[ib++];
   end

endmodule


////////////////////////////////////////////////////////////////////////////////
///  Problem 2
//
 ///  Modify batcher_merge so that it recursively implements a Batcher
 ///  odd/even merge module.
//
//     [✔] Recursively implement a Batcher Odd / Even merge module.
//
//     [✔] Assume that n is a power of 2.
//     [✔] Use sort2 to swap the values.
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Module must be synthesizable. Use command: genus -files syn.tcl
//
//     [✔] Use SimVision for debugging.
//     [✔] Modify testbench to facilitate solution ..
//         .. but code must pass original testbench.

module batcher_merge
  #( int n = 4, int w = 8 )
   ( output uwire [w-1:0] x[2*n], input uwire [w-1:0] a[n], b[n] );

   /// SOLUTION

   // Note: Input a and input b are each sorted.

   // Declare the outputs of the recursively instantiated merge modules.
   //
   uwire [w-1:0] xlo[n], xhi[n];

   if ( n == 1 ) begin

      // No need for recursion when each sorted sequence is one element.
      //
      assign xlo[0] = a[0];
      assign xhi[0] = b[0];

   end else begin

      localparam int nh = n/2;

      // Put even elements of a into ae ..
      // ..   odd elements of a into ao ..
      // .. and likewise for b.

      uwire [w-1:0] ae[nh], ao[nh], be[nh], bo[nh];

      for ( genvar i=0; i<nh; i++ )
```

```
        begin
           assign ae[i] = a[2*i];
           assign ao[i] = a[2*i+1];
           assign be[i] = b[2*i];
           assign bo[i] = b[2*i+1];
        end

       // Use one merge unit to merge the sorted sequences ae and bo ..
       //
       batcher_merge #(nh,w) mlo( xlo, ae, bo );
       //
       // and the other to merge sorted sequences ao and be.
       //
       batcher_merge #(nh,w) mhi( xhi, ao, be );
       //
       // This ensures that one of the two smallest elements is xlo[0] ..
       // .. and the other is xhi[0].

   end

   // Use 2-input sorters to complete the merge.
   //
   for ( genvar i=0; i<n; i++ )
     sort2 #(w) s2( x[2*i], x[2*i+1], xlo[i], xhi[i] );

endmodule


// Correctly functioning 2-input sorter.
module sort2
  #( int w = 8 )( output uwire [w-1:0] x0, x1, input uwire [w-1:0] a0, a1 );
   assign {x0, x1} = a0 <= a1 ? { a0, a1 } : { a1, a0 };
endmodule


////////////////////////////////////////////////////////////////////////////////
/// Testbench Code
//
//  The code below instantiates some of the modules above,
//  provides test inputs, and verifies the outputs.
//
//  The testbench may be modified to facilitate your solution. For
//  example, one might modify the testbench so that the first tests it
//  performs are those which make it easier to determine what the
//  problem is, for example, test inputs that are all 0's or all 1's.
//
//  Of course, the removal of tests which your module fails is not a
//  method of fixing a broken module.  The TA-bot will test your
//  code using a fresh copy of the testbench, not the one below.



// cadence translate_off

module sortx
  #( int n = 5,
     int modnum = 0,
     int mut_idx = 0,
     int w = 10,
     int max_muts = 3,
     int max_n = n)
   ( output uwire [w-1:0] xlong[max_muts][max_n],
     input uwire [w-1:0] a[n] );
```

```
   localparam int nlo = n/2;
   localparam int nhi = n - nlo;
   uwire [w-1:0] x[n];
   assign xlong[mut_idx][0:n-1] = x;
   uwire [w-1:0] alo[nlo] = a[0:nlo-1];
   uwire [w-1:0] ahi[nhi] = a[nlo:n-1];

   if ( modnum == 0 ) begin:A

      localparam string name = "Batcher Merge";
      localparam bit merge = 1;
      batcher_merge #(nlo,w) s(x,alo,ahi);

   end else if ( modnum == 1 ) begin:A

      localparam string name = "Batcher Sort";
      localparam bit merge = 0;
      batcher_sort #(n,w) s(x,a);

   end else if ( modnum == 2 ) begin:A

      localparam string name = "sort3";
      localparam bit merge = 0;

   end else begin:A

      localparam string name = "sort4";
      localparam bit merge = 0;

   end

endmodule


module testbench;

   localparam int w = 8;
   localparam int n_tests = 10;
   localparam int max_n = 32;
   localparam int max_muts = 12;

   logic [w-1:0] a[max_n];
   uwire [w-1:0] x[max_muts][max_n];

   typedef struct { int idx; string name; bit merge; int n; } Info;
   Info pi[$];

   for ( genvar i=0; i<2; i++ ) begin
      for ( genvar nlg = 1; nlg < 6; nlg++ ) begin
         localparam int n = 1 << nlg;
         localparam int idx = i * 6 + nlg;
         sortx #(n,i,idx,w,max_muts,max_n) s(x,a[0:n-1]);
         initial pi.push_back( '{ idx, s.A.name, s.A.merge, s.n } );
      end
   end

   initial begin

      automatic int g_elt_err_count = 0;
      automatic int g_sort_err_count = 0;

      $write("Starting testbench.\n");

      // Initialize the input to a recognizable pattern, which should
      // be overwritten but if not, we can tell. If we print the value in
```

```
      // hex.
      for ( int e = 0; e < max_n; e++ ) a[e] = 'haaaaaaaa;

      foreach ( pi[idx] ) begin

         automatic Info p = pi[idx];
         automatic string mut = p.name;
         automatic int n = p.n;
         automatic int s_size = n;
         automatic int nlo = n/2;
         automatic int nhi = n - nlo;
         automatic logic [w-1:0] shadow[] = new[s_size];
         automatic logic [w-1:0] alo[] = new[nlo];
         automatic logic [w-1:0] ahi[] = new[nhi];
         automatic int this_sort_err_count = 0;

         for ( int i = 0;  i < n_tests;  i++ ) begin

            automatic int this_elt_err_count = 0;

            // To make sure that the comparison is correct restrict the
            // key to a subset of bits.
            automatic int n_bits = {$random} % w + 1;
            automatic int mask = ( 1 << n_bits ) - 1;

            for ( int i=0; i<w; i++ ) begin
               automatic int b = {$random} % w;
               {mask[b],mask[i]} = {mask[i],mask[b]};
            end

            for ( int e = 0; e < s_size; e++ )
              begin
                 a[e] = {$random} & mask;
                 shadow[e] = a[e];
                 if ( e < nlo ) alo[e] = a[e]; else ahi[e-nlo] = a[e];
              end

            if ( p.merge ) begin
               alo.sort();
               ahi.sort();
               for ( int e=0; e<nlo; e++ ) a[e] = alo[e];
               for ( int e=nlo; e<n; e++ ) a[e] = ahi[e-nlo];
            end

            #1;

            shadow.sort();

            for ( int e = 0; e < s_size; e++ ) begin
               automatic logic [w-1:0] elt = x[p.idx][e];
               if ( shadow[e] === elt ) continue;
               this_elt_err_count++;
               g_elt_err_count++;
               if ( g_elt_err_count > 5 ) continue;
               $write
                 ("Mod %s, n=%0d, sort %2d idx %2d, wrong elt %d != %d (correct)\n",
                  mut, n, i, e, elt, shadow[e]);
            end

            if ( this_elt_err_count ) this_sort_err_count++;

         end

         if ( this_sort_err_count ) g_sort_err_count++;
```

```
            $write("Tests for %s (idx %0d)  n=%0d done, errors in %0d of %0d sorts.\n",
                   mut, p.idx, n,  this_sort_err_count, n_tests);

        end

        $write("Done with all tests, errors on %0d sorters.\n",
               g_sort_err_count);

    end

endmodule

// cadence translate_on
```

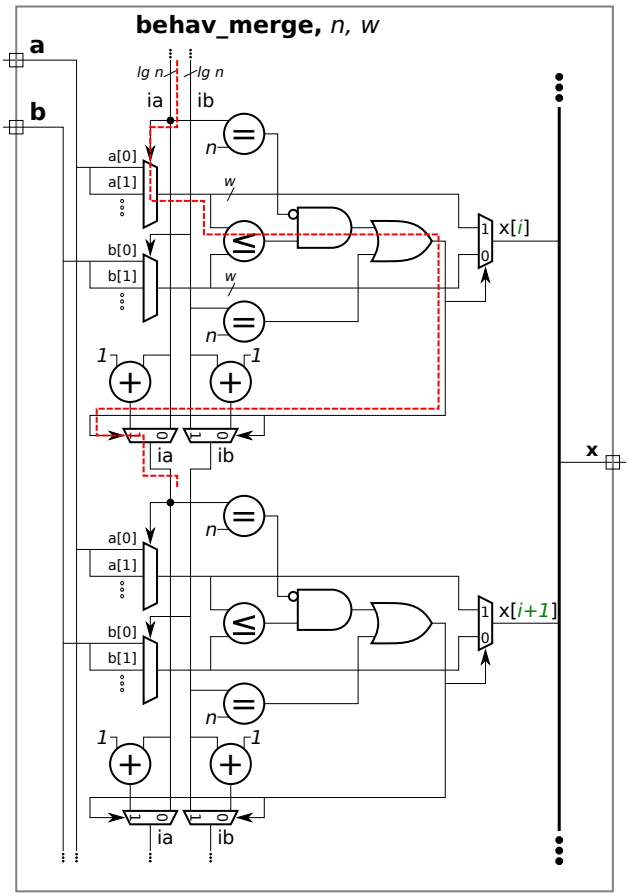
LSU EE 4755 Homework 6 Solution Due: 10 October 2018

Problem 1: Use the simple model to compute the cost and delay (critical path length) of the inferred hardware for module `behav_merge` from Homework 5. This module has two inputs, `a` and `b`, each of which is an n-element sorted sequence of w-bit unsigned integer values. Output `x` is a $2n$-element array of w-bit quantities. The module assigns elements of `a` and `b` to `x` so that `x` itself is a sorted sequence of the elements from `a` and `b`.

Show the cost and delay of `behav_merge` in terms of n and w. The Homework 5 module appears below. Use the tree implementation of multiplexors for cost and delay. (See the simple model notes.) Make reasonable optimizations, such as using the same multiplexor for `a[ia]` and `a[ia++]`. Avoid tedious optimizations such as varying the number of bits in `ia` and `ib`.

Solution on next page.

```
module behav_merge
  #( int n = 4, int w = 8 )
   ( output logic [w-1:0] x[2*n], input uwire [w-1:0] a[n], b[n] );

   logic [$clog2(n+1)-1:0] ia, ib;
   always_comb begin
      ia = 0; ib = 0;
      for ( int i = 0;  i < 2*n;  i++ )
        x[i] = ib == n || ia < n && a[ia] <= b[ib] ? a[ia++] : b[ib++];
   end
endmodule
```



The inferred hardware appears above. The problem did not explicitly ask for the inferred hardware, but cost and delay could not be found without it. The diagram shows the hardware resulting from two `for` loop iterations, for outputs `i` and `i+1`. The cost is dominated by the cost of the multiplexors implementing `a[ia]` and `b[ib]`. Each of these muxen, before optimization, has n inputs of w bits, for a cost of $3w(n-1)\,\mathrm{u_c}$ each. Since there are $2n$ iterations, the total cost of the `a` and `b` multiplexors will be $2n \times 2 \times 3w(n-1)\,\mathrm{u_c} \approx 12wn^2\,\mathrm{u_c}$. That's expensive. The cost will be less than that because the muxen for iteration $i < n$ only need i inputs. But accounting for that would not even cut the cost in half.

The muxen producing the value of `x[i]` cost $3w\,\mathrm{u_c}$ each for a total cost of $6wn\,\mathrm{u_c}$. The muxen passing `ia` and `ib` (incremented or not) cost $3\lg n\,\mathrm{u_c}$ each for a total cost of $12n\lg n\,\mathrm{u_c}$.

Magnitude comparison units ($\boxed{\leq}$) of w bits have a cost of $4w\,\mathrm{u_c}$ and a delay of $2w+1\,\mathrm{u_t}$, so the total cost of these units is $8wn\,\mathrm{u_c}$. The $\boxed{=n}$ *limit* units test whether `ia` and `ib` have reached their maximum value, `n`. In general an ω-bit comparison unit cost $4\omega - 1\,\mathrm{u_c}$ but in this case one input is a constant, and so the first column of

XOR gates is converted into either NOT gates or wire, and so the cost is reduced to $\omega - 1\,\mathrm{u_c}$. For `behav_merge` $\omega \to \lceil \lg(n+1) \rceil \approx \lg n$. There are two limit units per iteration, for a total of $4n$ units and so their total cost is $4n \lg n\,\mathrm{u_c}$.

The adders to increment `ia` and `ib` operate on $\lg n$-bit quantities. Unoptimized and based on a ripple implementation they would cost $9 \lg n\,\mathrm{u_c}$. But since one input is the constant 1 the ripple adder can be built using binary half-adders, at a cost of $3\,\mathrm{u_c}$ per bit, for a cost of $3 \lg n\,\mathrm{u_c}$. There are $4n$ adders for a total cost of $12n \lg n\,\mathrm{u_c}$.

The cost of everything is:

$$2n\Big[\overbrace{6w(n-1)}^{2\times \text{ big mux}} + \overbrace{3w}^{\texttt{x} \text{ mux}} + \overbrace{6\lg n}^{\texttt{iab} \text{ muxen}} + \overbrace{4w}^{\boxed{\leq}} + \overbrace{2\lg n}^{2\times\boxed{=n}} + \overbrace{6\lg n}^{2\times\boxed{+1}}\Big]\,\mathrm{u_c}$$

$$= 2n\big[6wn + w + 14\lg n\big]\,\mathrm{u_c}$$

The critical path is shown as a red dashed line. (The critical path also passes through `ib`, that's omitted for clarity and because those two paths are the same length.) Assuming a tree implementation for the mux and a ripple implementation for the comparison, each section has a critical path length of $((2\lg n) + 2w + 1 + 1 + 2)\,\mathrm{u_t}$. The total critical path length is $2n[(2\lg n) + 2w + 1 + 1 + 2]\,\mathrm{u_t} \approx (4n \lg n + 4nw)\,\mathrm{u_t}$. That's long. Even if the comparison used a tree-like design with a $\lg w$ delay the critical path through the merge unit would still be very long, at least compared to the Batcher odd/even merger.

Problem 2: As was probably mentioned, a proper n-element Batcher odd/even merge module is constructed from $\frac{n}{2}\lceil \lg n \rceil$ `sort2` modules, and the critical path length through a merge module is $\lceil \lg n \rceil$ `sort2` delays.

If the previous problem was solved correctly then the cost and critical path length of `behav_merge` should be much larger than a Batcher merge. But the behavioral code in `behav_merge` has a run time of $O(2n)$ running as an ordinary program, and consumes $O(2n)$ memory, both of which are optimal for an algorithm that must operate on all of $2n$ items. In fact, recursively applied code based on `behav_merge` can sort a sequence in $O(n \lg n)$ time, which is the best one can normally get in many cases.

What is it about the hardware realization of `behav_merge` that makes it so much less efficient than the software realization? Your answer should consider how much hardware is being used at each moment in time.

In the hardware version a piece of hardware is needed for each of the $2n$ outputs. That can't be avoided because this is combinational logic. So, for example, there are $2n$ $\boxed{\leq}$ comparison units, whereas in the execution of the software version there might be one ALU with just one comparison unit which gets used $2n$ times. This kind of efficiency could be realized with sequential logic.

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2018 Homework 7 -- SOLUTION
//
 /// Assignment https://www.ece.lsu.edu/koppel/v/2018/hw07.pdf

`default_nettype none

////////////////////////////////////////////////////////////////////////////////
///  Problem 1
//
 /// Complete mult_seq_ds_prob_1 as described in the handout and below.
//
//     [✔] Start multiplying when in_valid is 1 at a positive clock edge ..
//     [✔] .. even if that means abandoning a multiplication in progress.
//     [✔] Set out_avail to 1 when prod holds the result for
//         most recent plier*cand.
//
//     [✔] The module must pass the testbench.
//         Average cycles should be w/m+1
//     [✔] The module must be synthesizable.
//     [✔] Make sure that synthesized hardware is reasonably fast.
//
//     [✔] Code must be reasonably efficient.
//     [✔] Do not change module parameters.
//     [✔] Do not change ports, EXCEPT changing between var and net kinds.
//     [✔] Don't assume that parameter values will match those used here.
//     [✔] USE DEBUGGING TOOLS LIKE SimVision.
//

module mult_seq_ds_prob_1
  #( int w = 16, int m = 2 )
   ( output logic [2*w-1:0] prod,
     // SOLUTION: Change kind of out_avail from net (uwire) to var.
     output var logic  out_avail,
     input uwire clk, in_valid,
     input uwire [w-1:0] plier, cand );

   localparam int iterations = ( w + m - 1 ) / m;
   localparam int iter_lg = $clog2(iterations);
   localparam logic [w+m-1:0] zero = 0;  // Used to set precision to w+m bits.

   uwire [iterations-1:0][m-1:0] cand_2d = cand;

   bit [iter_lg:0] iter;
   logic [2*w-1:0] accum;

   always_ff @( posedge clk ) begin

      /// SOLUTION, Problem 1
      //
      //  - Start a new multiplication whenever in_valid is 1.
      //  - When multiplication is finished set out_avail to 1.
      //
      if ( in_valid ) begin

         // If in_valid is 1 start a multiplication.

         accum = cand;
         iter = 0;
         out_avail = 0;

      end else if ( !out_avail && iter == iterations ) begin

         // If a multiplication is in progress (!out_avail) ..
         // .. and we just finished the last iteration of a multiplication ..
         // .. make the result available.

         out_avail = 1;
         prod = accum;

      end

      // Add on a partial product.
      // Do this whether or not a multiplication is in progress.

      accum = { zero + plier * accum[m-1:0] + accum[2*w-1:w], accum[w-1:m] };
      iter++;

   end

endmodule
```

```
////////////////////////////////////////////////////////////////////////////////
///  Problem 2
//
 /// Complete mult_seq_d_prob_2 as described in the handout and below.
//
//     [✔] Skip over multiplicand digits that are zero.
//     [✔] Start multiplying when in_valid is 1 at a positive clock edge ..
//     [✔] .. even if that means abandoning a multiplication in progress.
//     [✔] Set out_avail to 1 when prod holds the result for
//         most recent plier*cand.
//
//     [✔] The module must pass the testbench.
//         Average cycles should be less than w/m+1
//     [✔] The module must be synthesizable.
//         The period should not be too much longer than the original module.
//     [✔] Make sure that synthesized hardware is reasonably fast.
//
//     [✔] The module must be synthesizable.
//     [✔] Code must be reasonably efficient.
//     [✔] Do not change module parameters.
//     [✔] Do not change ports, EXCEPT changing between var and net kinds.
//     [✔] Don't assume that parameter values will match those used here.
//     [✔] USE DEBUGGING TOOLS LIKE SimVision.

module mult_seq_d_prob_2
  #( int w = 16, int m = 2 )
   ( output logic [2*w-1:0] prod,
     // SOLUTION: Change kind of out_avail from net (uwire) to var.
     output logic out_avail,
     input uwire clk, in_valid,
     input uwire [w-1:0] plier, cand );

   localparam int iterations = ( w + m - 1 ) / m;
   localparam int iter_lg = $clog2(iterations);

   uwire [iterations-1:0][m-1:0] cand_2d = cand;

   bit [iter_lg-1:0] iter;
   logic [2*w-1:0] accum;

   always_ff @( posedge clk ) begin

      logic [iter_lg-1:0] next_iter;

      /// SOLUTION -- Problem 2
      //
      //  Implement handshaking.
      //  Computation is completed when iter is zero. (See below.)
      //
      if ( in_valid ) begin

         iter = 0;
         accum = 0;
         out_avail = 0;

      end else if ( !out_avail && iter == 0 ) begin

         prod = accum;
         out_avail = 1;

      end

      accum += plier * cand_2d[iter] << ( iter * m );

      /// SOLUTION -- Problem 2
      //
      //  Set iter to ..
      //  .. index of next non-zero multiplicand digit ..
      //  .. or to zero if multiplication is complete.
      //
      //  Scan multiplicand digits starting at most significant digit.
      //  Update next_iter whenever ..
      //   i > iter   ( meaning that that partial product not yet use ) ..
      //   and digit, cand_2d[i], is non-zero.
      //
      next_iter = 0;
      for ( int i=iterations-1;  i>0;  i-- )
        if ( i>iter && cand_2d[i] ) next_iter = i;
      iter = next_iter;

   end

endmodule


////////////////////////////////////////////////////////////////////////////////
```

/// Comparison Modules
///

/// **The modules below are for reference.**

```
module mult_seq_ds_prob_1_orig
  #( int w = 16, int m = 2 )
   ( output logic [2*w-1:0] prod,
     output uwire out_avail,
     input uwire clk, in_valid,
     input uwire [w-1:0] plier, cand );

   /// DO NOT MODIFY THIS MODULE.
   //  It is to be used for comparison when performing synthesis.

   localparam int iterations = ( w + m - 1 ) / m;
   localparam int iter_lg = $clog2(iterations);
   localparam logic [w+m-1:0] zero = 0;  // Used to set precision to w+m bits.

   uwire [iterations-1:0][m-1:0] cand_2d = cand;

   bit [iter_lg:0] iter;
   logic [2*w-1:0] accum;

   always_ff @( posedge clk ) begin

      if ( iter == iterations ) begin

         prod = accum;
         accum = cand;
         iter = 0;

      end

      // Note: accum[m-1:0] is the same as cand_2d[iter];

      accum = { zero + plier * accum[m-1:0] + accum[2*w-1:w], accum[w-1:m] };
      iter++;

   end

endmodule

module mult_seq_d_prob_2_orig
  #( int w = 16, int m = 2 )
   ( output logic [2*w-1:0] prod,
     output uwire out_avail,
     input uwire clk, in_valid,
     input uwire [w-1:0] plier, cand );

   /// DO NOT MODIFY THIS MODULE.
   //  It is to be used for comparison when performing synthesis.

   localparam int iterations = ( w + m - 1 ) / m;
   localparam int iter_lg = $clog2(iterations);

   uwire [iterations-1:0][m-1:0] cand_2d = cand;

   bit [iter_lg:0] iter;
   logic [2*w-1:0] accum;

   always_ff @( posedge clk ) begin

      if ( iter == iterations ) begin

         prod = accum;
         accum = 0;
         iter = 0;

      end

      accum += plier * cand_2d[iter] << ( iter * m );

      iter++;

   end

endmodule
```

//
/// Testbench Code

```
// cadence translate_off
```

```
program reactivate
   (output uwire clk_reactive, output int cycle_reactive,
    input uwire clk, input var int cycle);
   assign clk_reactive = clk;
   assign cycle_reactive = cycle;
endprogram

module testbench;

   localparam int w = 20;
   localparam int num_tests = 400;
   localparam int NUM_MULT = 20;
   localparam int err_limit = 7;

   bit use_others;
   logic [w-1:0] plier, cand;
   logic [w-1:0] plierp[NUM_MULT], candp[NUM_MULT];
   logic [2*w-1:0] prod[NUM_MULT];
   uwire availn[NUM_MULT];
   logic avail[NUM_MULT];
   logic in_valid[NUM_MULT];

   typedef struct { int tidx; int cycle_start; } Test_Vector;

   typedef struct { int idx;
                    int err_count = 0;
                    int err_timing = 0;
                    Test_Vector tests_active[$];
                    bit all_tests_started = 0;
                    bit seq = 0; bit pipe = 0;
                    bit bpipe = 0;
                    int deg = 1;
                    int ncompleted = 0;
                    int cyc_tot = 0;
                    int latency = 0;
                    } Info;
   Info pi[string];

   localparam int cycle_limit = num_tests * w * 4;
   int cycle;
   bit done;
   logic clock;

   logic clk_reactive;
   int cycle_reactive;
   reactivate ra(clk_reactive,cycle_reactive,clock,cycle);


   initial begin
      clock = 0;
      cycle = 0;

      fork
         forever #10 cycle += clock++;
         wait( done );
         wait( cycle >= cycle_limit )
           $write("*** Cycle limit exceeded, ending.\n");
      join_any;

      $finish();
   end

   task pi_seq(input int idx, input string name, input int deg);
      automatic string m = $sformatf("%s Deg %0d", name, deg);
      pi[m].deg = deg;
      pi[m].idx = idx; pi[m].seq = 1; pi[m].bpipe = 0;
   endtask

   task pi_bseq(input int idx, input string name, input int deg);
      automatic string m = $sformatf("%s Deg %0d", name, deg);
      pi[m].deg = deg;
      pi[m].idx = idx; pi[m].seq = 1; pi[m].bpipe = 1;
   endtask

   task pi_pipe(input int idx, input string name, input int deg);
      automatic string m = $sformatf("%s Deg %0d", name, deg);
      pi[m].deg = deg;
      pi[m].idx = idx; pi[m].seq = 1; pi[m].pipe = 1; pi[m].bpipe = 0;
   endtask
   task pi_bpipe(input int idx, input string name, input int deg);
      automatic string m = $sformatf("%s Deg %0d", name, deg);
      pi[m].deg = deg;
      pi[m].idx = idx; pi[m].seq = 1; pi[m].pipe = 1; pi[m].bpipe = 1;
   endtask

   mult_seq_ds_prob_1 #(w,1) prob1_m1(prod[6], availn[6], clock,
```

```
                        in_valid[6], plierp[6], candp[6]);
initial pi_bseq(6,"Prob 1",prob1_m1.m);

mult_seq_ds_prob_1 #(w,2) prob1_m2(prod[7], availn[7], clock,
                        in_valid[7], plierp[7], candp[7]);
initial pi_bseq(7,"Prob 1",prob1_m2.m);

mult_seq_ds_prob_1 #(w,4) prob1_m4(prod[9], availn[9], clock,
                        in_valid[9], plierp[9], candp[9]);
initial pi_bseq(9,"Prob 1",prob1_m4.m);

mult_seq_ds_prob_1_orig #(w,1) ms14(prod[14], availn[14], clock,
                        in_valid[14], plierp[14], candp[14]);
initial pi_seq(14,"Seq",ms14.m);

mult_seq_ds_prob_1_orig #(w,2) ms4(prod[4], availn[4], clock,
                        in_valid[4], plierp[4], candp[4]);
initial pi_seq(4,"Seq",ms4.m);

mult_seq_ds_prob_1_orig #(w,4) ms5(prod[5], availn[5], clock,
                        in_valid[5], plierp[5], candp[5]);
initial pi_seq(5,"Seq",ms5.m);


mult_seq_d_prob_2 #(w,1) prob2_m1(prod[17], availn[17], clock,
                        in_valid[17], plierp[17], candp[17]);
initial pi_bseq(17,"Prob 2",prob2_m1.m);

mult_seq_d_prob_2 #(w,2) prob2_m2(prod[16], availn[16], clock,
                        in_valid[16], plierp[16], candp[16]);
initial pi_bseq(16,"Prob 2",prob2_m2.m);

mult_seq_d_prob_2 #(w,4) prob2_m4(prod[15], availn[15], clock,
                        in_valid[15], plierp[15], candp[15]);
initial pi_bseq(15,"Prob 2",prob2_m4.m);

always @* begin

   foreach ( availn[i] ) begin
      if ( availn[i] !== 1'bz ) avail[i] = availn[i];
   end

end

// Array of multiplier/multiplicand values to try out.
// After these values are used a random number generator will be used.
//
int tests[$] = {1,1, 1,2, 1,3, 1,4, 1,5,  1,32,  32, 1};

initial begin

   automatic int awaiting = pi.size();

   logic [w-1:0] pliers[num_tests], cands[num_tests];

   done = 0;

   foreach ( pi[mut] ) begin
      automatic int midx = pi[mut].idx;
      automatic int steps = ( w + pi[mut].deg - 1 ) / pi[mut].deg;
      automatic int latency =
        !pi[mut].seq ? 1 : !pi[mut].pipe ? 2 * steps : steps;
      pi[mut].latency = latency;
      if ( pi[mut].bpipe == 0 ) begin
         avail[midx] = 1;
      end
      in_valid[midx] = 0;
   end

   for ( int i=0; i<num_tests; i++ ) begin

      automatic int num_bits_c = {$random()}%w + 1;
      automatic logic [w-1:0] mask_c = ( (w+1)'(1) << num_bits_c ) - 1;
      automatic int num_bits_p = {$random()}%w + 1;
      automatic logic [w-1:0] mask_p = ( (w+1)'(1) << num_bits_p ) - 1;

      pliers[i] = tests.size() ? tests.pop_front() : {$random()}&mask_p;
      cands[i] = tests.size() ? tests.pop_front() : {$random()}&mask_c;

   end

   fork begin
      forever @( negedge clk_reactive ) begin
         foreach ( pi[mut] ) begin
            automatic int midx = pi[mut].idx;
            if ( !in_valid[midx] && pi[mut].pipe ) begin
               plierp[midx] = cycle;
```

```
            candp[midx] = 1;
          end
        end
      end
    end join_none;

    repeat ( 2 * w ) @( negedge clock );

    foreach ( pi[mutii] ) begin
      automatic string muti = mutii;

      fork begin
        automatic string mut = muti;
        automatic int midx = pi[mut].idx;
        for ( int i=0; i<num_tests; i++ ) begin
          automatic int gap_cyc =
            !pi[mut].pipe ? w * 2 :
            ( {$random} % 2 ) ? {$random} % ( w + 2 ) : 0;
          automatic Test_Vector tv;
          repeat ( gap_cyc ) @( negedge clock );
          plierp[midx] = pliers[i];
          candp[midx] = cands[i];
          in_valid[midx] = 1;
          tv.tidx = i;
          tv.cycle_start = cycle;
          pi[mut].tests_active.push_back( tv );
          @( negedge clock );
          in_valid[midx] = 0;
        end
        pi[mut].all_tests_started = 1;
      end join_none;

      fork begin
        automatic string mut = muti;
        automatic int midx = pi[mut].idx;
        while ( 1 ) begin
          @( negedge clock );
          while ( pi[mut].tests_active.size() == 0
                  && !pi[mut].all_tests_started )
            @( negedge clock );
          if ( pi[mut].tests_active.size() == 0 ) break;
          begin
            automatic Test_Vector tv = pi[mut].tests_active.pop_front();
            automatic int i = tv.tidx;
            automatic logic [2*w-1:0] shadow_prod = pliers[i] * cands[i];
            automatic int eta = tv.cycle_start + pi[mut].latency;
            automatic bit timing_err = 0;
            automatic int delta_t;
            if ( pi[mut].bpipe ) begin

              if ( !pi[mut].pipe && cycle == tv.cycle_start )
                @( negedge clock );

              while ( !avail[midx] && cycle < eta ) @( negedge clock );
              if ( !avail[midx] || cycle > eta ) begin
                timing_err = 1;
                if ( pi[mut].err_timing++ < err_limit )
                  $write("At cyc %4d (eta %0d) avail not set for %s (idx %0d) after %0d cycles for 0x%0h*0x%0h.\n",
                         cycle, eta, mut, midx, cycle - tv.cycle_start,
                         pliers[i], cands[i]);
              end
            end else begin
              wait ( cycle >= eta );
            end
            delta_t = cycle - tv.cycle_start;
            if ( !timing_err ) begin
              pi[mut].ncompleted++;
              pi[mut].cyc_tot += delta_t;
            end
            if ( !timing_err && shadow_prod !== prod[midx] ) begin
              pi[mut].err_count++;
              if ( pi[mut].err_count < err_limit ) begin
                $write
                  ("%-15s test %5d  cyc %0d+%0d (%0d) wrong: 0x%0h * 0x%0h:  0x%0h != 0x%0h (correct)\n",
                   mut, i, tv.cycle_start, delta_t, pi[mut].latency,
                   pliers[i], cands[i],
                   prod[midx], shadow_prod);
              end
            end
          end
        end
        awaiting--;
      end join_none;

    end

    wait( awaiting == 0 || cycle > cycle_limit );
```

```
      $write("At cycle %0d.  Error types:  couldn't test / wrong result / timing\n",cycle);

      foreach ( pi[ mut ] )
        $write("For %-18s ran %4d tests, %4d/%4d/%4d errors found. Avg cyc %.1f\n",
               mut, num_tests,
               num_tests - pi[mut].ncompleted,
               pi[mut].err_count, pi[mut].err_timing,
               pi[mut].seq ? real'(pi[mut].cyc_tot) / pi[mut].ncompleted : 1);

      done = 1;
      $write("Modules instantiated with w = %0d.\n",w);

      $finish(2);

   end

endmodule

// cadence translate_on
```

LSU EE 4755 Homework 8 Solution Due: 27 November 2018

Problem 1: Appearing below is the output of the simulator and synthesis script, showing data for the Homework 7 solution modules. Modules are simulated and synthesized for $w = 32$.

```
Module Name                              Area  Period  Period
                                               Target  Actual
mult_seq_ds_prob_1_w32_m1              157813    1000   14926
mult_seq_ds_prob_1_w32_m2              185493    1000   15431
mult_seq_ds_prob_1_w32_m4              242568    1000   16296

mult_seq_d_prob_2_w32_m1               288580    1000   31944
mult_seq_d_prob_2_w32_m2               301203    1000   32204
mult_seq_d_prob_2_w32_m4               329226    1000   32192

For Prob 1 Deg 1       ran  400 tests,    0/   0/   0 errors found. Avg cyc 33.0
For Prob 1 Deg 2       ran  400 tests,    0/   0/   0 errors found. Avg cyc 17.0
For Prob 1 Deg 4       ran  400 tests,    0/   0/   0 errors found. Avg cyc 9.0

For Prob 2 Deg 1       ran  400 tests,    0/   0/   0 errors found. Avg cyc 9.5
For Prob 2 Deg 2       ran  400 tests,    0/   0/   0 errors found. Avg cyc 7.3
For Prob 2 Deg 4       ran  400 tests,    0/   0/   0 errors found. Avg cyc 5.0
Modules instantiated with w = 32.
```

The Problem 1 modules are based on the streamlined multiplier and so are faster. But the Problem 2 modules skip zeros. Based on the data above, indicate the ways, if any, that the Problem 2 modules are better than the Problem 1 modules. Explain using the numbers above.

By skipping zeros the Problem 2 modules should compute a result with lower latency (in less time) than the Problem 1 modules, which require $\lceil w/m \rceil + 1$ cycles regardless of the numbers being multiplied. The latency for a multiplication is the product of the clock period and the average number of cycles required. For the Problem 1 modules that works out to

$$33 \times 14.926\,\text{ns} = 492.6\,\text{ns}, \quad 17 \times 15.431\,\text{ns} = 262.3\,\text{ns}, \text{ and } \quad 9 \times 16.296\,\text{ns} = 146.7\,\text{ns}$$

for the degree (m) 1, 2, and 4 modules respectively. Though the clock periods for the Problem 2 modules are larger, fewer cycles are needed to produce an answer according to the data collected by the testbench. (See the number to the right of `Avg cyc`.) The Problem 2 module latencies are

$$9.5 \times 31.944\,\text{ns} = 303.5\,\text{ns}, \quad 7.3 \times 32.204\,\text{ns} = 235.1\,\text{ns}, \text{ and } \quad 5.0 \times 32.192\,\text{ns} = 161.0\,\text{ns}.$$

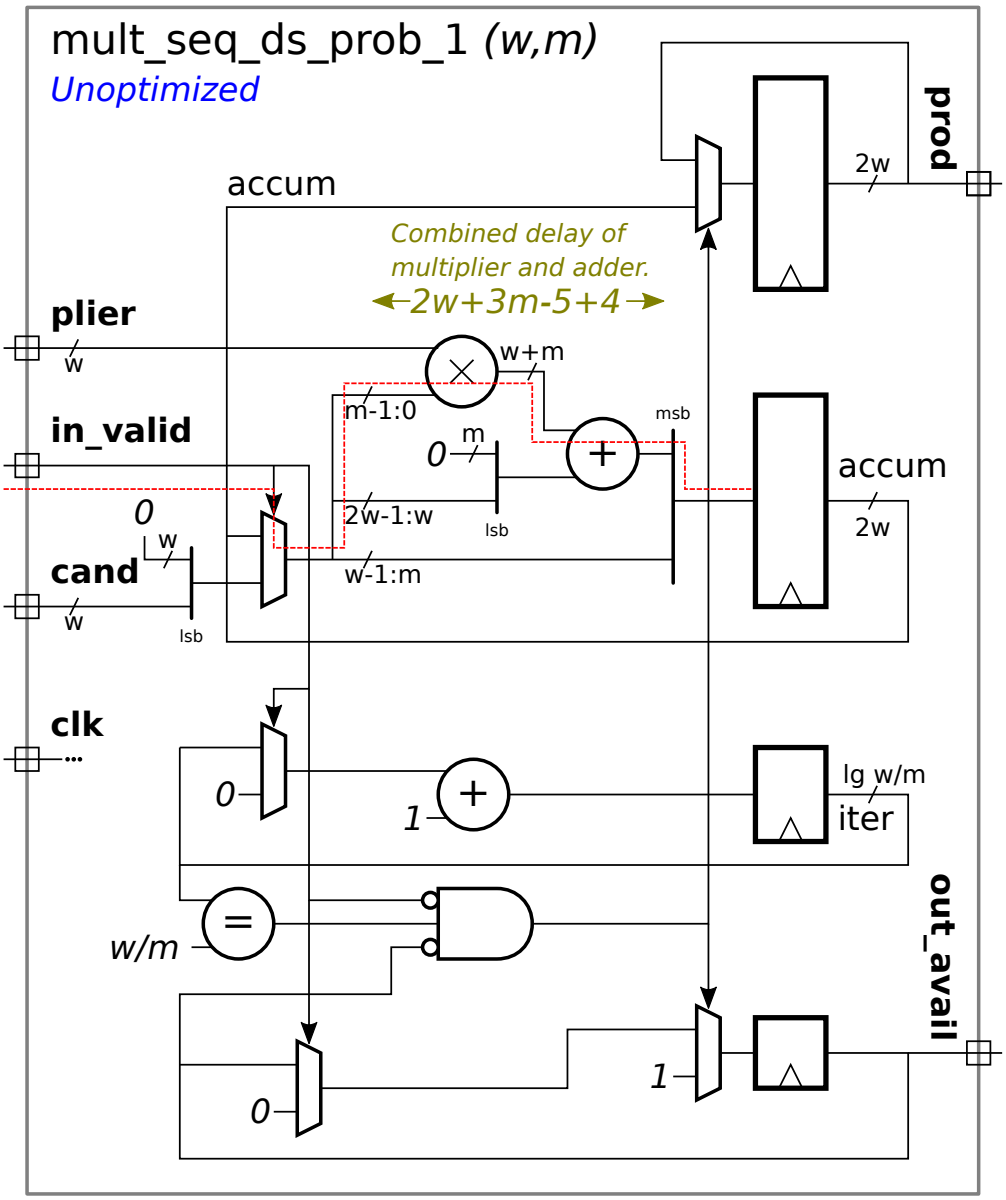
In all but the $m = 4$ case the Problem 2 module has a lower latency than the respective Problem 1 module.

There are more problems on the next pages.

Problem 2: Appearing below is a solution to Homework 7, Problem 1, the streamlined degree-m multiplier with handshaking. The complete solution is at `https://www.ece.lsu.edu/koppel/v/2018/hw07-sol.v.html`. For this problem assume that w and m are both powers of 2.

```
module mult_seq_ds_prob_1 #( int w = 16, int m = 2 )
   ( output logic [2*w-1:0] prod,  output logic  out_avail,
     input uwire clk, in_valid,    input uwire [w-1:0] plier, cand );

   localparam int iterations = ( w + m - 1 ) / m;
   localparam int iter_lg = $clog2(iterations);

   uwire [iterations-1:0][m-1:0] cand_2d = cand;

   bit [iter_lg:0] iter;
   logic [2*w-1:0] accum;

   always_ff @( posedge clk ) begin

      if ( in_valid ) begin

         accum = cand;
         iter = 0;
         out_avail = 0;

      end else if ( !out_avail && iter == iterations ) begin

         out_avail = 1;
         prod = accum;

      end

      accum = { 0 + plier * accum[m-1:0] + accum[2*w-1:w], accum[w-1:m] };
      iter++;
   end

endmodule
```

(*a*) Show the hardware that will be inferred for this module. The Inkscape SVG format diagram of the hardware for the streamlined sequential module from the class demo notes can be used as a starting point. It is at `https://www.ece.lsu.edu/koppel/v/2018/ill-mul-seq-str.svg`.



Solution appears above with the critical path shown in red. The hardware is un-optimized. Optimization opportunities include the logic for computing `out_avail`.

(*b*) Compute the cost and delays for this module using the simple model. Show these in terms of w and m. Clearly show the critical path on your diagram.

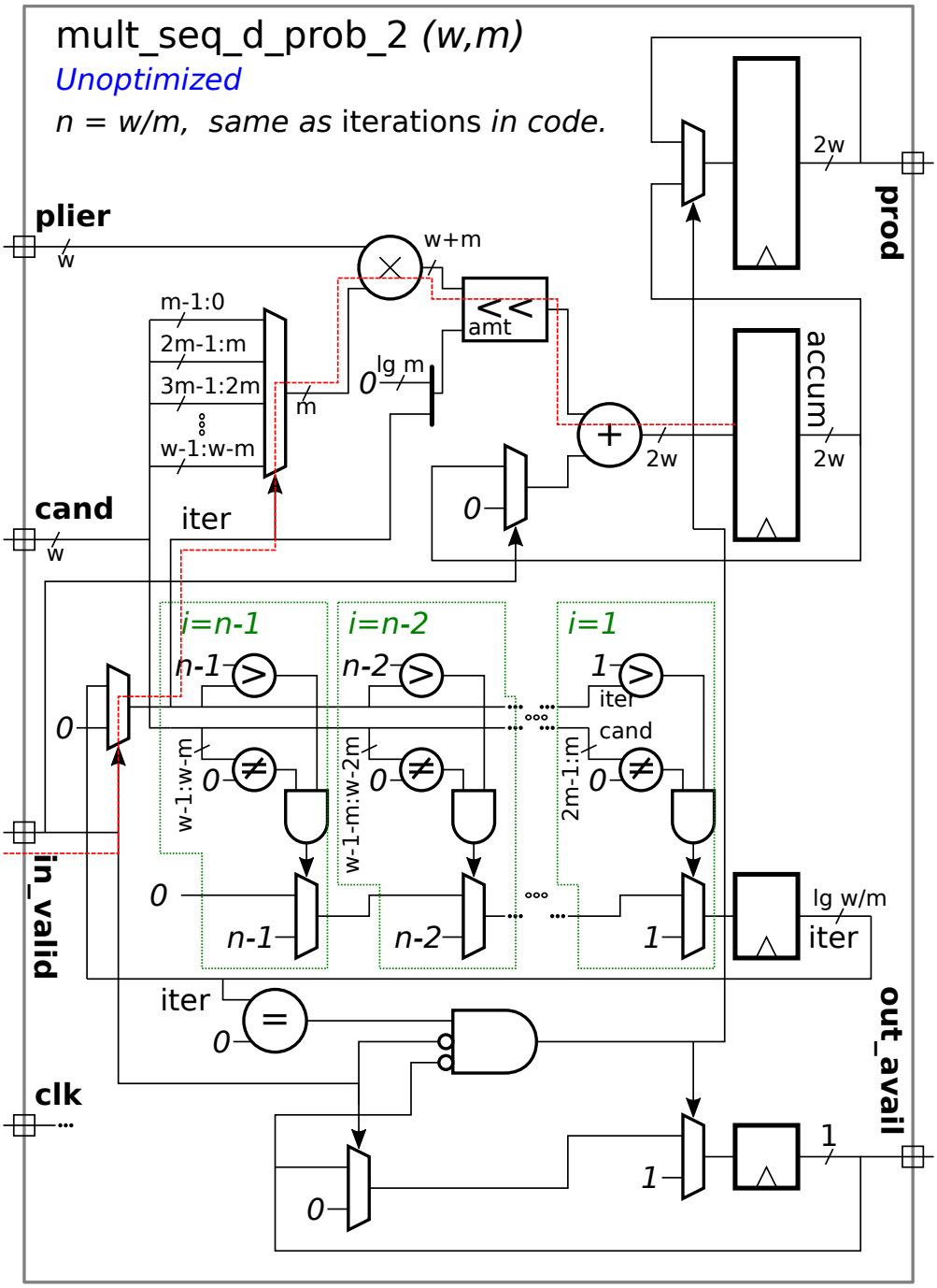
See the solution to Problem 3 for a complete delay and timing analysis. In this (Problem 2) module the cost of the adder is less because it is $w + m$ bits, rather than $2w$ bits for the Problem 3 adder. Also, this module does not use a shifter or a mux to extract the multiplicand bits.

There is a problem on the next page.

Problem 3: Appearing below is a solution to Homework 7, Problem 2, the streamlined degree-m multiplier with handshaking. The complete solution is at `https://www.ece.lsu.edu/koppel/v/2018/hw07-sol.v.html`. For this problem assume that w and m are both powers of 2.

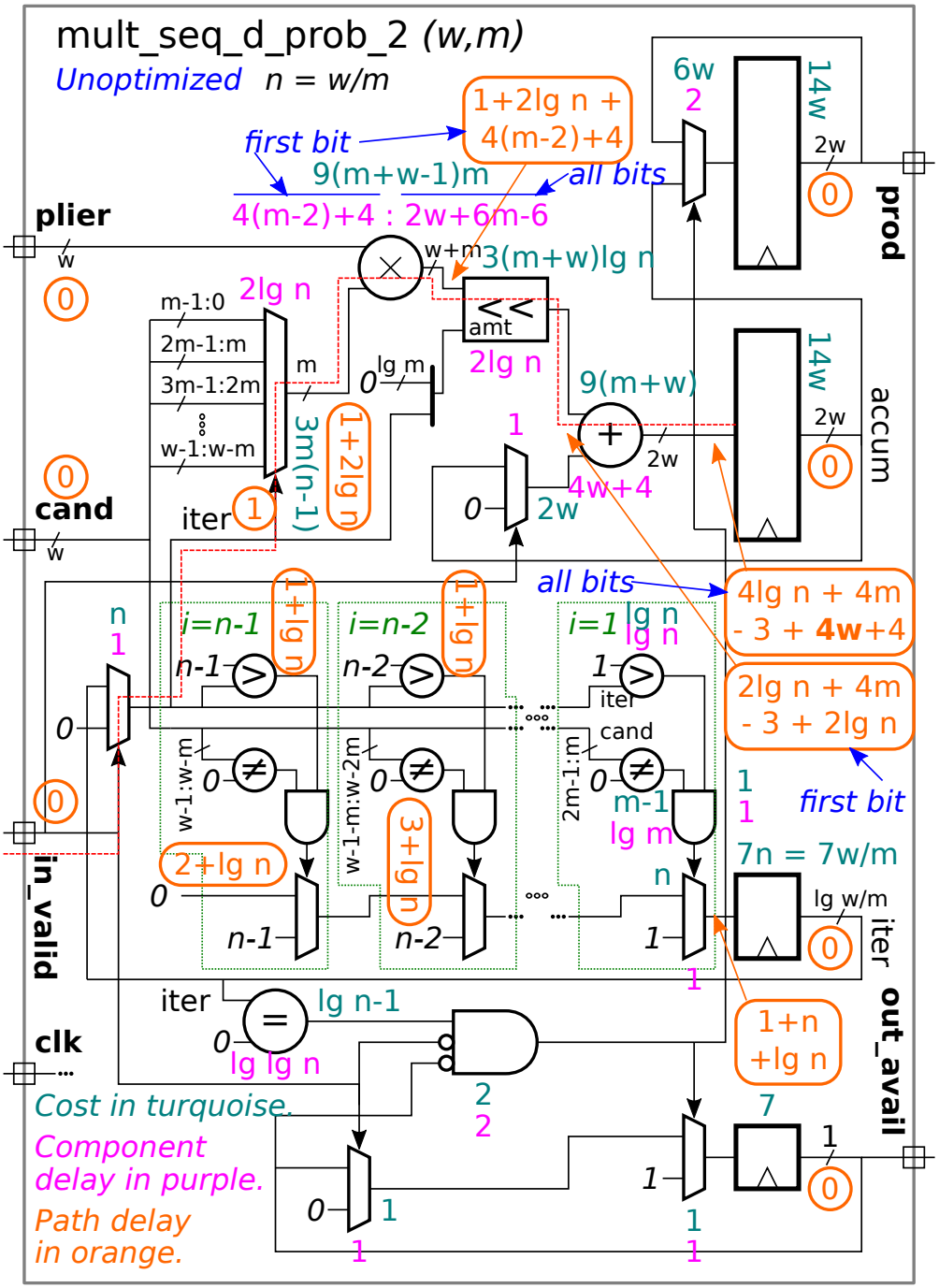
```
module mult_seq_d_prob_2 #( int w = 16, int m = 2 )
   ( output logic [2*w-1:0] prod,   output logic out_avail,
     input uwire clk, in_valid,     input uwire [w-1:0] plier, cand );

   localparam int iterations = ( w + m - 1 ) / m;
   localparam int iter_lg = $clog2(iterations);

   uwire [iterations-1:0][m-1:0] cand_2d = cand;

   bit [iter_lg-1:0] iter;
   logic [2*w-1:0] accum;

   always_ff @( posedge clk ) begin

      logic [iter_lg-1:0] next_iter;

      if ( in_valid ) begin
         iter = 0;
         accum = 0;
         out_avail = 0;
      end else if ( !out_avail && iter == 0 ) begin
         prod = accum;
         out_avail = 1;
      end

      accum += plier * cand_2d[iter] << ( iter * m );

      next_iter = 0;
      for ( int i=iterations-1;  i>0;  i-- )
        if ( i>iter && cand_2d[i] ) next_iter = i;
      iter = next_iter;
   end

endmodule
```

(*a*) Show the hardware that will be inferred for this module.



Hardware shown above with the critical path shown in red.

(*b*) Compute the cost and delays for this module using the simple model. Show these in terms of w and m. Clearly show the critical path on your diagram.



The costs and delay of each component are shown in the diagram above. The path delay for selected paths is shown in the circled orange numbers. Note that one input to all of the comparison units (for example, the zero in $\neq 0$), is a constant, reducing their costs and delays. Many of the multiplexors also have one constant data input.

The interesting thing to compare is the time needed to compute the updated `accum` value versus the time needed to find the next non-zero digit. The $i >$ iter comparison, because i is a constant, takes time $\lg w/m\ \mathrm{u_t} = \lg n\ \mathrm{u_t}$ and

the $\neq 0$ takes less, especially if $w/m > m$. The mux delay is $1\,\mathrm{u_t}$ because one data input is a constant. The time to generate the new `iter` signal is $(1 + n + \lg n)\,\mathrm{u_t}$.

The updated `accum` value consumes most of the time. Inputs arrive at the multiplier at time $1 + 2\lg n$. For an unoptimized m-bit by $w + m$-bit multiplier, the least significant bit takes $(4(m-2)+4)\,\mathrm{u_t}$ to compute. Since the shifter can shift by n possible amounts its delay is $2\lg n$. The least significant bit arrives at the adder at time $1 + 2\lg n + 4(m-2) + 4 + 2\lg n = (4\lg n + 4m - 3)\,\mathrm{u_t}$ (see the diagram). The adder requires $(4w+4)\,\mathrm{u_t}$ to finish and so the adder output is ready at time $(4\lg n + 4m - 3 + 4w + 4)\,\mathrm{u_t}$.

The clock period would include six more cycles for the latch setup time.

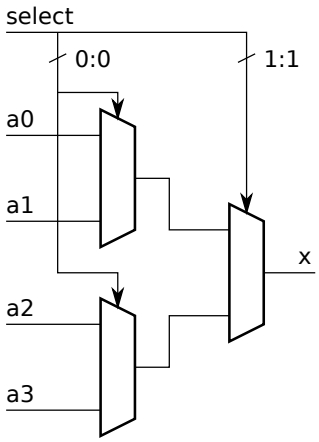
21 Fall 2017 Solutions

LSU EE 4755 Homework 1 Solution Due: 8 September 2017

Start working on the solutions to the problems below on paper, but complete them using the computers in the lab. For instructions visit `https://www.ece.lsu.edu/koppel/v/proc.html`. *For the complete Verilog for this assignment without visiting the lab visit* `https://www.ece.lsu.edu/koppel/v/2017/hw01.v.html`.

Problem 1: Appearing below, and in `hw01.v`, is a Verilog description of a 2-input multiplexer, `mux2`, and a partially completed description of a 4-input mux, `mux4`, along with a diagram showing how a four-input mux can be made using three two-input multiplexers. Complete `mux4` as described in the diagram.

It is important that `mux4` instantiate three `mux2` modules. Other correct 4-input multiplexer implementations will not receive credit. Also, don't forget to set the parameters correctly when instantiating modules.




```
module mux2
  #( int w = 16 )
   ( output uwire [w-1:0] x,
     input uwire s,
     input uwire [w-1:0] a, b );

   assign  x = s == 0 ? a : b;

endmodule

module mux4
  #( int w = 6 )
   ( output uwire [w-1:0] x,
     input uwire [1:0] s,
     input uwire [w-1:0] a[3:0] );

   /// SOLUTION
   //
   //  Notice that wires and modules are named based upon the select
   //  bits for which they connect to the output.
   //
   uwire [w-1:0] x0x, x1x;

   mux2 #(w) m0x(x0x, s[0], a[0], a[1]);
   mux2 #(w) m1x(x1x, s[0], a[2], a[3]);

   mux2 #(w) mxx(x, s[1], x0x, x1x);

endmodule
```

Problem 2: Appearing below is a `mux8` module. Complete `mux8` so that it implements an 8-input multiplexer using two `mux4` modules and one `mux2` module. Notice that the data input to `mux8` is an 8-element array of w-bit quantities. To see how to extract a subrange of an array (called a *part select* in Verilog) see the `testbench` module. Solve this problem by generalizing the technique appearing in the previous problem.

Credit will only be given for `mux8` modules that *instantiate* two `mux4` modules and a `mux2` module. Yes, `assign x = a[s];` is correct and the best way to do it in other situations, but the goal here is to learn about instantiation.

```
module mux8
  #( int w = 5 )
   ( output uwire [w-1:0] x,
     input uwire [2:0] s,
     input uwire [w-1:0] a[7:0] );

   /// SOLUTION
   uwire [w-1:0] x0xx, x1xx;

   mux4 #(w) m0xx(x0xx, s[1:0], a[3:0]);
   mux4 #(w) m1xx(x1xx, s[1:0], a[7:4]);

   mux2 #(w) m(x, s[2], x0xx, x1xx);

endmodule
```

Appearing below is the start of the testbench code. To see the complete testbench and other modules follow `https://www.ece.lsu.edu/koppel/v/2017/hw01.v.html`.

```
module testbench();

   localparam int w = 10;
   localparam int n_in_max = 8;
   localparam int n_mut = 3;

   uwire [w-1:0] x[n_mut];
   logic [2:0]  s;
   logic [w-1:0] a[n_in_max-1:0];

   mux2 #(w) mm2(x[0], s[0], a[0], a[1]);
   mux4 #(w) mm4(x[1], s[1:0], a[3:0]);
   mux8 #(w) mm8(x[2], s[2:0], a[7:0]);

   initial begin

      automatic int n_test = 0;
      automatic int n_err = 0;
```

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2017 Homework 1
//
 /// SOLUTION


`default_nettype none

////////////////////////////////////////////////////////////////////////////////
///  Problem 1
//
 /// Modify mux4 so that it implements a 4-input mux as described in handout.
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Code must instantiate three mux2 modules as shown in hw01.pdf.
//     [✔] Make sure that parameters set correctly in instantiation.

module mux4
  #( int w = 6 )
   ( output uwire [w-1:0] x,
     input uwire [1:0] s,
     input uwire [w-1:0] a[3:0] );

   /// SOLUTION
   //
   //  Notice that wires and modules are named based upon the select
   //  bits for which they connect to the output.
   //
   uwire [w-1:0] x0x, x1x;

   mux2 #(w) m0x(x0x, s[0], a[0], a[1]);
   mux2 #(w) m1x(x1x, s[0], a[2], a[3]);

   mux2 #(w) mxx(x, s[1], x0x, x1x);


endmodule

module mux2
  #( int w = 16 )
   ( output uwire [w-1:0] x,
     input uwire s,
     input uwire [w-1:0] a, b );

   assign  x = s == 0 ? a : b;

endmodule


////////////////////////////////////////////////////////////////////////////////
///  Problem 2
//
 /// Modify mux8 so that it implements an 8-input mux as described in handout.
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Code must instantiate two mux4 and one mux2 modules.
//     [✔] Make sure that parameters set correctly in instantiation.


module mux8
  #( int w = 5 )
```

```
   ( output uwire [w-1:0] x,
     input uwire [2:0] s,
     input uwire [w-1:0] a[7:0] );

   /// SOLUTION

   uwire [w-1:0] x0xx, x1xx;

   mux4 #(w) m0xx(x0xx, s[1:0], a[3:0]);
   mux4 #(w) m1xx(x1xx, s[1:0], a[7:4]);

   mux2 #(w) m(x, s[2], x0xx, x1xx);


endmodule


////////////////////////////////////////////////////////////////////////////////
/// Testbench Code
//
//  The code below instantiates some of the modules above,
//  provides test inputs, and verifies the outputs.
//
//  The testbench may be modified to facilitate your solution. Of
//  course, the removal of tests which your module fails is not a
//  method of fixing a broken module. (One might modify the testbench
//  so that the first tests it performs are those which make it easier
//  to determine what the problem is, for example, test inputs that
//  are all 0's or all 1's.)



// cadence translate_off


module testbench();

   localparam int w = 10;
   localparam int n_in_max = 8;
   localparam int n_mut = 3;

   uwire [w-1:0] x[n_mut];
   logic [2:0]  s;
   logic [w-1:0] a[n_in_max-1:0];

   mux2 #(w) mm2(x[0], s[0], a[0], a[1]);
   mux4 #(w) mm4(x[1], s[1:0], a[3:0]);
   mux8 #(w) mm8(x[2], s[2:0], a[7:0]);

   initial begin

      automatic int n_test = 0;
      automatic int n_err = 0;

      for ( int i=0; i < n_in_max; i++ ) begin
         n_test++;
         s = i;
         for ( int j=0; j<n_in_max; j++ ) a[j] = $random;
         #1;
         for ( int m=0; m<n_mut; m++ ) begin

            automatic int n_in = 2 << m;
            automatic int sm = i & ( n_in - 1 );
```

```
            if ( x[m] !== a[sm] ) begin
               n_err++;
               $write("Error in %0d-input mux for s=%0d, 0x%0x != 0x%0x (correct)\n",
                      n_in, sm, x[m], a[sm]);
            end
         end
      end
      $write("Done with %0d tests, %0d errors found.\n",n_test,n_err);
   end

endmodule

// cadence translate_on
```

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2017 Homework 2 -- SOLUTION
//

 /// Assignment http://www.ece.lsu.edu/koppel/v/2017/hw02.pdf

////////////////////////////////////////////////////////////////////////////////
/// Problem 1 -- SOLUTION
//
 /// Modify interp so that it performs linear interpolation. See the handout
 /// and module interp_behav.
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Module must be synthesizable.
//     [✔] Module must do some FP arithmetic.
//     [✔] Modify include statements (at end) for any new ChipWare modules.

`default_nettype none

module interp
  #( int jw = 12, int amax = 255 )
   ( output uwire valid,
     output uwire [7:0] aj,
     input uwire [31:0] x1, a1, x2, a2,
     input uwire [jw-1:0] j );

   /// Port Data Representations
   //
   //  Inputs:
   //    x1,a1,x2,a2: shortreal.
   //    j:           Unsigned integer.
   //
   //  Outputs:
   //    valid:       Boolean, 0 = no, 1 = yes.
   //    aj:          Unsigned integer.
   //

   localparam logic [2:0] rnd_even = 3'b000; // Round to closest. Default.

   uwire [jw:0] x1i, x2i;

   /// SOLUTION

   /// First, generate the valid signal.

   // Convert x1 and x2 to integers.
   //
   fp_ftoi #( jw+1 ) ftoi1(x1i, x1);
   fp_ftoi #( jw+1 ) ftoi2(x2i, x2);
   //
   // Note: Since the ChipWare float-to-int module can only convert to
   // a signed integer and x is unsigned need to make the integer one
   // bit wider to accommodate the sign bit that we won't need.
   // Otherwise, values >= 2^{jw-1}, for the default, 2^11 = 2048,
   // will be clamped to the maximum 12-bit signed representation,
   // 2047.

   // Check whether j is between x1 and x2.
   //
```

```
   assign valid = x1i + j <= x2i;

   //
   /// Perform the interpolation: aj = a1 + j * ( a2 - a1 ) / ( x2 - x1 )
   //

   uwire [31:0] delta_x, delta_a, dadx, jr, jdadx, ajr;
   uwire [7:0] status[2]; // Unused status connections for CW modules.

   fp_sub sdx(delta_x, x2, x1);
   fp_sub sda(delta_a, a2, a1);

   CW_fp_div div
     ( .status(status[0]), .z(dadx), .a(delta_a), .b(delta_x), .rnd(rnd_even) );

   fp_itof #(jw) itof(jr,j);
   //
   // Note: Module performs an unsigned conversion, so we don't need to
   // widen j by one bit. See ftoi3 below and ftoi1 and ftoi2 above.

   CW_fp_mult mul
     ( .status(status[1]), .z(jdadx), .a(jr), .b(dadx), .rnd(rnd_even) );

   fp_add add(ajr,a1,jdadx);

   /// Convert the interpolated value to an integer and clamp it between
   // 0 and amax.

   // Declare aji signed so that the comparison operator works correctly
   // for aji < 0.
   //
   uwire signed [8:0] aji;

   fp_ftoi #( 9 ) ftoi3( aji, ajr );

   assign     aj = aji < 0 ? 0 : aji > amax ? amax : aji[7:0];
   //
   // Note that when amax is 255 the clamp isn't necessary
   // because the float-to-int module clamps to the maximum representable
   // value, which is 255 for a 9-bit signed integer.

endmodule


module fp_itof
  #( int wid = 10, logic i_is_signed = 0 )
   ( output uwire [31:0] f, input uwire [wid-1:0] i);

   uwire [7:0] status;
   localparam logic [2:0] rnd_even = 3'b000;

   CW_fp_i2flt #( .isize(wid), .isign(i_is_signed) )
     itof ( .status(status), .a(i), .z(f), .rnd(rnd_even) );
endmodule



////////////////////////////////////////////////////////////////////////////////
/// Convenience wrappers around ChipWare modules.
///
 //  Feel free to define additional modules.
```

```
//  See http://www.ece.lsu.edu/v/ref.html for ChipWare documentation.

module fp_add(output uwire [31:0] x, input uwire [31:0] a, b );
   uwire [7:0] status;
   localparam logic [2:0] rnd_even = 3'b000; // Round to closest. Default.
   CW_fp_add add( .status(status), .z(x), .a(a), .b(b), .rnd(rnd_even) );
endmodule

module fp_sub(output uwire [31:0] x, input uwire [31:0] a, b );
   uwire [7:0] status;
   localparam logic [2:0] rnd_even = 3'b000; // Round to closest. Default.
   CW_fp_sub sub( .status(status), .z(x), .a(a), .b(b), .rnd(rnd_even) );
endmodule

module fp_ftoi
  #( int wid = 10 )
   ( output uwire [wid-1:0] i, input uwire [31:0] f);

   uwire [7:0] status;
   localparam logic [2:0] rnd_even = 3'b000; // Round to closer integer.
   localparam logic [2:0] rnd_trun = 3'b001; // Round towards zero. (truncate)
   localparam logic [2:0] rnd_minf = 3'b011; // Round towards -infinity.

   CW_fp_flt2i #( .isize(wid) ) ftoi
     ( .status(status), .z(i), .a(f), .rnd(rnd_trun) );
endmodule

////////////////////////////////////////////////////////////////////////////////
/// Behavioral Interpolation Module
//
//  Module below is correct but not synthesizable.


// cadence translate_off

module interp_behav
  #( int jw = 12,
     int amax = 255 )
   ( output logic valid,
     output logic [7:0] aj,
     input uwire [31:0] x1, a1, x2, a2,
     input uwire [jw-1:0] j );

   always_comb begin

      automatic shortreal x1r = $bitstoshortreal(x1);
      automatic shortreal x2r = $bitstoshortreal(x2);
      automatic shortreal a1r = $bitstoshortreal(a1);
      automatic shortreal a2r = $bitstoshortreal(a2);

      automatic int x1i = $floor(x1r);
      automatic int x2i = $floor(x2r);
      automatic int xj = x1i + j;

      shortreal dadx, ajr;

      valid = xj <= x2i;

      dadx = ( a2r - a1r ) / ( x2r - x1r );
      ajr = a1r + j * dadx;
      aj = ajr < 0 ? 0 : ajr > amax ? amax : $floor(ajr);
```

```
   end

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Testbench Code
//
//  The code below instantiates some of the modules above,
//  provides test inputs, and verifies the outputs.
//
//  The testbench may be modified to facilitate your solution. Of
//  course, the removal of tests which your module fails is not a
//  method of fixing a broken module. (One might modify the testbench
//  so that the first tests it performs are those which make it easier
//  to determine what the problem is, for example, test inputs that
//  are all 0's or all 1's.)


module testbench();

   localparam bit trunc_x1 = 1;

   localparam int err_max_display = 20;
   localparam shortreal tolerance = 0.0001;

   localparam int num_tests = 2000;
   localparam int xmin = 0;
   localparam int xmax = 3839;
   localparam longint rand_max = longint'(1) << 32;
   localparam shortreal xscale = shortreal'(xmax) / rand_max;
   localparam shortreal short_len = 5;
   localparam shortreal short_scale = short_len / rand_max;

   localparam int amax = 255;
   localparam shortreal ascale = shortreal'(amax) / rand_max;

   localparam int jw = 12;

   typedef struct
     {
      string name;
      int err_valid = 0;
      int err_aj = 0;
      } Info;
   Info muts[int];
   task new_interp(input int idx, input string name);
      muts[idx].name = name;
   endtask

   localparam int mut_n_max = 5;

   logic [jw-1:0] mj;
   uwire        mvalid[mut_n_max];
   uwire [7:0] maj[mut_n_max];
   logic [31:0] mx1, mx2, ma1, ma2;

   interp_behav #(jw) i0(mvalid[0], maj[0], mx1, ma1, mx2, ma2, mj);
   initial new_interp(0,"interp_behav");
   interp #(jw) i1(mvalid[1], maj[1], mx1, ma1, mx2, ma2, mj);
```

```
initial new_interp(1,"interp");

initial begin

   for ( int i=0; i<num_tests; i++ ) begin

      automatic bit short_line = $random & 1;
      automatic shortreal x[] = { {$random} * xscale, {$random} * xscale };
      shortreal len1;
      shortreal x1, x2, a1, a2, dadx;
      int x1i, x2i;
      int npts;
      x.sort();
      len1 = x[1] - x[0];
      if ( short_line && len1 > short_len )
        x[1] = x[0] + {$random} * short_scale;

      if ( trunc_x1 ) x[0] = $floor(x[0]);
      x1 = x[0];  x2 = x[1];
      mx1 = $shortrealtobits(x1);
      mx2 = $shortrealtobits(x2);

      a1 = {$random} * ascale;
      a2 = {$random} * ascale;
      ma1 = $shortrealtobits(a1);
      ma2 = $shortrealtobits(a2);

      dadx = ( a2 - a1 ) / ( x2 - x1 );

      x1i = $floor(x1);
      x2i = $floor(x2);
      npts = x2i - x1i + 1;

      for ( int j=0; j<npts+10; j++ ) begin

         automatic shortreal aj = a1 + ( x1i + j - x1 ) * dadx;
         automatic int aji = aj < 0 ? 0 : aj > amax ? amax : $floor(aj);
         automatic shortreal ajfrac = aj - aji;
         automatic int tol =
           ajfrac < tolerance ? -1 : ajfrac > 1 - tolerance ? 1 : 0;
         automatic int ajalt = aji + tol;
         automatic logic valid = j < npts;
         mj = j;

         #1;

         foreach ( muts[m] ) begin

            if ( mvalid[m] !== valid ) begin
               if ( muts[m].err_valid < err_max_display )
                 $write("Err in %s for %4.1f, %4.1f, j=%0d, valid %0d != %0d (correct)\n",
                        muts[m].name, x1, x2, j, mvalid[m], valid );
               muts[m].err_valid++;
            end
            if ( valid && mvalid[m] && maj[m] !== aji && maj[m] !== ajalt )
              begin
                 if ( muts[m].err_aj < err_max_display )
                   $write("Err in %s for %4.1f, %4.1f, j=%0d, aj=%.4f  %0d != %0d (correct)\n",
                          muts[m].name, a1, a2, j, aj, maj[m], aji );
                 muts[m].err_aj++;
              end
         end
```

```
          end

      end

      foreach ( muts[m] )
        $write("Done with tests for %s, %0d + %0d errors.\n",
             muts[m].name,muts[m].err_valid, muts[m].err_aj);

   end

endmodule

`define SIMULATION_ON

// cadence translate_on

`default_nettype wire

`ifdef SIMULATION_ON

`include "/apps/linux/cadence/DDIEXPORT23/GENUS231/share/synth/lib/chipware/sim/verilog/CW/CW_fp_mult.v"
`include "/apps/linux/cadence/DDIEXPORT23/GENUS231/share/synth/lib/chipware/sim/verilog/CW/CW_fp_add.v"
`include "/apps/linux/cadence/DDIEXPORT23/GENUS231/share/synth/lib/chipware/sim/verilog/CW/CW_fp_sub.v"
`include "/apps/linux/cadence/DDIEXPORT23/GENUS231/share/synth/lib/chipware/sim/verilog/CW/CW_fp_div.v"
`include "/apps/linux/cadence/DDIEXPORT23/GENUS231/share/synth/lib/chipware/sim/verilog/CW/CW_fp_i2flt.v"
`include "/apps/linux/cadence/DDIEXPORT23/GENUS231/share/synth/lib/chipware/sim/verilog/CW/CW_fp_flt2i.v"

`else

`include "/apps/linux/cadence/DDIEXPORT23/GENUS231/share/synth/lib/chipware/syn/CW/CW_fp_mult.v"
`include "/apps/linux/cadence/DDIEXPORT23/GENUS231/share/synth/lib/chipware/syn/CW/CW_fp_add.v"
`include "/apps/linux/cadence/DDIEXPORT23/GENUS231/share/synth/lib/chipware/syn/CW/CW_fp_sub.v"
`include "/apps/linux/cadence/DDIEXPORT23/GENUS231/share/synth/lib/chipware/syn/CW/CW_fp_i2flt.v"
`include "/apps/linux/cadence/DDIEXPORT23/GENUS231/share/synth/lib/chipware/syn/CW/CW_fp_flt2i.v"
`include "/apps/linux/cadence/DDIEXPORT23/GENUS231/share/synth/lib/chipware/syn/CW/CW_fp_div.v"

`endif
```

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2017 Homework 4
//
 /// SOLUTION

 /// Assignment http://www.ece.lsu.edu/koppel/v/2017/hw04.pdf


 /// Additional Resources
  //
  // Verilog Documentation
  //    The Verilog Standard
  //      http://standards.ieee.org/getieee/1800/download/1800-2012.pdf
  //    Introductory Treatment (Warning: Does not include SystemVerilog)
  //      Brown & Vranesic, Fundamentals of Digital Logic with Verilog, 3rd Ed.
  //
  // Account Setup and Emacs (Text Editor) Instructions
  //      http://www.ece.lsu.edu/koppel/v/proc.html
  //      To learn Emacs look for Emacs tutorial.


`default_nettype none

////////////////////////////////////////////////////////////////////////////////
/// Problem 1
//
 /// Modify maxrun so that it keeps track of the current and maximum runs.
//
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Module must be synthesizable.
//     [✔] Code must be reasonably efficient.


 /// Solution 1: Written for maximum code clarity.
//
module maxrun
  #( int w = 2,
     int c = 4 )
   ( output uwire [w-1:0] len,
     output logic [c-1:0] mr_char,
     input uwire clk, reset, mr,
     input uwire [c-1:0] in_char );

   logic [w-1:0] cr_len, mr_len;
   logic [c-1:0] prev_char;

   assign len = mr ? mr_len : cr_len;

   always_ff @( posedge clk ) begin

      if ( reset ) mr_len = 0;

      if ( !reset && in_char == prev_char )
        cr_len++;
      else
        cr_len = 1;

      if ( cr_len > mr_len )
        begin
           mr_len = cr_len;
           mr_char = in_char;
        end
```

```
      prev_char = in_char;

   end

endmodule

 /// Solution 2: Written for high performance.
//
module maxrun_opt
  #( int w = 2,
     int c = 4 )
   ( output uwire [w-1:0] len,
     output logic [c-1:0] mr_char,
     input uwire clk, reset, mr,
     input uwire [c-1:0] in_char );

   logic [w-1:0] cr_len, mr_len;
   logic [c-1:0] prev_char;

   assign len = mr ? mr_len : cr_len;

   always_ff @( posedge clk ) begin

      logic match;
      match = in_char == prev_char;

      /// Approach to Reducing Critical Path
      //
      //   To keep addition off the critical path ..
      //   .. check cr_len >= mr_len && match ..
      //   .. rather than using incremented cr_len for: cr_len > mr_len.
      //
      //   Based on experimentation, use ..
      //   .. cr_len >= mr_len ..
      //   .. instead of ..
      //   .. cr_len == mr_len ..
      //   .. even though cr_len == mr_len is easier to compute.

      if ( reset ) begin
         mr_len = 1;
         mr_char = in_char;
      end else if ( cr_len >= mr_len && match ) begin
         mr_len = cr_len + 1;
         mr_char = in_char;
      end

      if ( !reset && match )
        cr_len = cr_len + 1;
      else
        cr_len = 1;

      prev_char = in_char;

   end

endmodule

////////////////////////////////////////////////////////////////////////////////
/// Testbench Code
//
//  The code below instantiates some of the modules above,
//  provides test inputs, and verifies the outputs.
//
```

```
//  The testbench may be modified to facilitate your solution. Of
//  course, the removal of tests which your module fails is not a
//  method of fixing a broken module. (One might modify the testbench
//  so that the first tests it performs are those which make it easier
//  to determine what the problem is, for example, test inputs that
//  are all 0's or all 1's.)


// cadence translate_off

program reactivate
   (output uwire clk_reactive, output int cycle_reactive,
    input uwire clk, input var int cycle);
   assign clk_reactive = clk;
   assign cycle_reactive = cycle;
endprogram

module testbench;

   localparam int char_wid = 8;
   localparam int count_wid = 10;

   localparam int test_num_chars = 100;
   localparam int cycle_limit = test_num_chars + 20;

   localparam int nmuts = 1;

   localparam int char_mask = ( 1 << char_wid ) - 1;

   uwire [count_wid-1:0] len[nmuts];
   uwire [char_wid-1:0] mr_char[nmuts];
   logic [char_wid-1:0] char, shadow_last_char;
   logic mr;

   logic clock, reset;
   bit done;
   int cycle;

   logic clk_reactive;
   int cycle_reactive;
   reactivate ra(clk_reactive,cycle_reactive,clock,cycle);

   initial begin
      clock = 0;
      cycle = 0;

      fork
         forever #10 cycle += clock++;
         wait( done );
         wait( cycle >= cycle_limit )
           $write("*** Cycle limit exceeded, ending.\n");
      join_any;

      $finish();
   end

   maxrun_opt #(count_wid,char_wid) mr1 (len[0],mr_char[0],clock,reset,mr,char);

   initial begin

      automatic int n_err_cr_len = 0, n_err_mr_len = 0, n_err_mr_char = 0;
      int shadow_mr_len, shadow_mr_char, shadow_cr_len;
      bit is_err_cr_len, is_err_mr_len, is_err_mr_char;

      done = 0;
```

```
      reset = 0;
      char = 0;
      mr = 0;

      @( posedge clk_reactive );

      for ( int i=0; i<test_num_chars; i++ ) begin

         automatic bit do_reset = i == 0 || {$random} % 10 == 0;
         automatic bit do_new_char = {$random} % 3 == 0;
         logic [count_wid-1:0] mr_len, cr_len;

         @( negedge clock );

         shadow_last_char = char;

         if ( do_new_char ) char = {$random} & char_mask;
         reset = do_reset;

         if ( !do_reset && char === shadow_last_char )
           shadow_cr_len++;
         else
           shadow_cr_len = 1;

         if ( do_reset )
            shadow_mr_len = 0;

         if ( shadow_cr_len > shadow_mr_len ) begin
            shadow_mr_len = shadow_cr_len;
            shadow_mr_char = char;
         end

         @( posedge clk_reactive );

         repeat ( 2 ) begin
            if ( mr ) mr_len = len[0]; else cr_len = len[0];
            mr = !mr;
            #0; #0;
         end

         is_err_cr_len = shadow_cr_len !== cr_len;
         is_err_mr_len = shadow_mr_len !== mr_len;
         is_err_mr_char = shadow_mr_char !== mr_char[0];

         $write
           ("%5d %1s c=%2x    cr_len %3d %s    mr_len %3d %s    mr_c %2x %s\n",
            i, do_reset ? "r" : " ", char,
            cr_len,
            is_err_cr_len ? $sformatf("!= %3d", shadow_cr_len) : "ok    ",
            mr_len,
            is_err_mr_len ? $sformatf("!= %3d", shadow_mr_len) : "ok    ",
            mr_char[0],
            is_err_mr_char ? $sformatf("!= %2x", shadow_mr_char) : "ok   " );

         if ( shadow_cr_len !== cr_len ) n_err_cr_len++;
         if ( shadow_mr_len !== mr_len ) n_err_mr_len++;
         if ( shadow_mr_char !== mr_char[0] ) n_err_mr_char++;

      end

      $write("Done with %0d tests, %0d %0d %0d errors found.\n",
             test_num_chars,
             n_err_cr_len,
             n_err_mr_len,
             n_err_mr_char);
```

```
      done = 1;
   end
endmodule
// cadence translate_on
```

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2017 Homework 5 -- SOLUTION
//
 /// Assignment http://www.ece.lsu.edu/koppel/v/2017/hw05.pdf

`default_nettype none

////////////////////////////////////////////////////////////////////////////////
///  Problem 1
//
 /// Complete so that lookup_char finds index of character.
//
//     [✔] Module must be synthesizable.
//     [✔] Code must be reasonably efficient.
//     [✔] Do not change module parameters.
//     [✔] Do not change ports, EXCEPT changing between var and net kinds.
//     [✔] The module must synthesize into combinational logic (no latches).
//     [✔] Don't assume that parameter values will match those used here.
//     [✔] See a 2016 homework assignment.

module lookup_char
  #( int w = 4,
     int n = 3,
     logic [w-1:0] chars[n] = '{ "a", "2", "g" },
     int c = $clog2(n) )
   ( output logic found,
     output logic [c-1:0] idx,
     input uwire [w-1:0] char );

   always_comb begin
      found = 0;
      idx = 0;
      for ( int i=0; i<n; i++ )
        if ( chars[i] == char ) begin found = 1;  idx = i; end
   end

endmodule

////////////////////////////////////////////////////////////////////////////////
///  Problem 2
//
 /// Complete so that nest checks for properly nested characters.
//
//     [✔] Use lookup_char in nest.
//     [✔] Module must be synthesizable.
//     [✔] Code must be reasonably efficient.
//     [✔] Do not change module parameters.
//     [✔] Do not change ports, EXCEPT changing between var and net kinds.
//     [✔] Outputs bad, level, and awaiting should change on positive clk edge.
//     [✔] Don't assume that parameter values will match those used here.


module nest
  #( int d = 8,
     int w = 8,
     int n = 2,
     logic [w-1:0] char_open[n] = { 1, 2 },
     logic [w-1:0] char_close[n] = { 3, 4 },
     int dw = $clog2(d+1) )
  ( output logic [dw-1:0] level,
    output uwire [w-1:0] awaiting,
    output uwire is_open, is_close,
    output logic bad,
    input uwire clk, reset,
    input uwire [w-1:0] in_char );

   localparam int nw = $clog2(n);

   uwire [nw-1:0] loidx, lcidx;

   lookup_char #(w,n,char_open) l1(is_open,loidx,in_char);
   lookup_char #(w,n,char_close) l2(is_close,lcidx,in_char);

   logic [nw-1:0] stack [1:d];

   assign awaiting = char_close[stack[level]];

   always_ff @( posedge clk ) begin

      if ( reset ) begin

         level = 0;
         bad = 0;
```

```
      end else begin

         if ( is_open ) begin

            if ( level == d ) bad = 1;
            level++;
            stack[level] = loidx;

         end else if ( is_close ) begin

            if ( awaiting != in_char || !level ) bad = 1;
            level--;

         end

      end

   end

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Testbench Code
//
//  The code below instantiates some of the modules above,
//  provides test inputs, and verifies the outputs.
//
//  The testbench may be modified to facilitate your solution. Of
//  course, the removal of tests which your module fails is not a
//  method of fixing a broken module. (One might modify the testbench
//  so that the first tests it performs are those which make it easier
//  to determine what the problem is, for example, test inputs that
//  are all 0's or all 1's.)


// cadence translate_off

program reactivate
   (output uwire clk_reactive, output int cycle_reactive,
    input uwire clk, input var int cycle);
   assign clk_reactive = clk;
   assign cycle_reactive = cycle;
endprogram


module testbench;

   localparam int w = 8;
   localparam int max_depth = 6;
   localparam int dw = $clog2(max_depth);

   // Maximum number of groups for which to show traces.
   //
   localparam int show_groups_bad = 3;
   localparam int show_groups_good = 2;

   localparam int num_seq = 1000;

   localparam int cycle_limit = num_seq * 1000;

   localparam logic [w-1:0] char_open[] =  { "(", "[", "{", "<" };
   localparam logic [w-1:0] char_close[] = { ")", "]", "}", ">" };
   localparam int num_pairs = 4;

   initial begin
      if ( num_pairs != char_open.size() )
        $error("Size of char_open, %0d, different than num_pairs., %0d",
               char_open.size(), num_pairs);
   end

   uwire is_op, is_cl, bad;
   logic [w-1:0] in_char;
   uwire [w-1:0] await;
   logic [dw-1:0] lev;

   logic clock, reset;
   bit done;
   int cycle;

   logic clk_reactive;
   int cycle_reactive;
   reactivate ra(clk_reactive,cycle_reactive,clock,cycle);

   int num_tests, errs_bad, errs_op, errs_cl, errs_lv, errs_await;
```

```
   initial begin

      clock = 0;
      cycle = 0;

      fork
         forever #10 cycle += clock++;
         wait( done );
         wait( cycle >= cycle_limit )
           $write("*** Cycle limit exceeded, ending.\n");
      join_any;

      $write
        ("End of %0d tests, errors: %0d + %0d + %0d + %0d + %0d = %0d\n",
         num_tests,
         errs_op, errs_cl, errs_bad, errs_lv, errs_await,
         errs_op + errs_cl + errs_bad + errs_lv + errs_await );

      $finish();
   end


   nest #(max_depth,w,num_pairs,char_open,char_close)
   n1(lev, await, is_op, is_cl, bad, clock, reset, in_char );

   localparam string oe[] = '{"  ","er"};
   logic [w-1:0] chars_plain[$];
   bit chars_br[int];
   int nchars_plain;

   initial begin

      automatic int groups_good_count = 0;
      automatic int groups_bad_count = 0;

      num_tests = 0;
      errs_bad = 0;
      errs_op = 0;
      errs_cl = 0;
      errs_lv = 0;
      errs_await = 0;

      foreach ( char_open[c] ) chars_br[c] = 1;
      foreach ( char_close[c] ) chars_br[c] = 1;
      for ( int i=0; i<26; i++ ) begin
         chars_plain.push_back("A" + i);
         chars_plain.push_back("a" + i);
      end

      nchars_plain = chars_plain.size();

      done = 0;
      reset = 0;
      in_char = 0;

      @( negedge clk_reactive );

      for ( int s=0; s<num_seq; s++ ) begin

         automatic int targ_depth = {$random} % max_depth;
         automatic int curr_depth = 0;
         automatic int stack[$];
         automatic bit hit_target = 0;
         automatic bit back_to_0 = 0;
         automatic int c = 0;
         automatic bit shadow_bad = 0;
         automatic bit botch_close = {$random} % 2;
         automatic int bad_cyc = 0;
         automatic byte shadow_await;
         automatic string trace_text[$];
         automatic int some_err = 0;
         automatic bit err_op;
         automatic bit err_cl;

         trace_text.push_back("\n");

         reset = 1;
         @( negedge clock );
         @( negedge clock );
         reset = 0;

         while ( !back_to_0 && c < 100 && bad_cyc < 3 ) begin

            automatic bit plain = {$random} & 1;

            automatic bit b_open
              = {$random} & 'hff > ( hit_target ? 'hc0 : 'h40 );
```

```
      if ( plain ) begin

         in_char = chars_plain[ {$random} % nchars_plain ];

      end else begin

         automatic int idx = {$random} % num_pairs;

         if ( b_open ) begin

            in_char = char_open[ idx ];
            curr_depth++;
            stack.push_back(idx);
            if ( curr_depth == targ_depth ) hit_target = 1;
            if ( curr_depth > max_depth ) shadow_bad = 1;

         end else begin

            automatic bit botch_this_close
              = botch_close && {$random} & 'hff > 'h40;
            automatic int tos = curr_depth > 0 ? stack.pop_back() : idx;
            in_char
              = char_close[ botch_this_close ? (tos+1)%num_pairs : tos ];
            if ( curr_depth == 0 || botch_this_close ) shadow_bad = 1;
            curr_depth--;
            if ( curr_depth == 0 && hit_target ) back_to_0 = 1;

         end
      end

      shadow_await = char_close[stack.size() ? stack[stack.size()-1] : 0];

      #1;

      err_op = is_op !== ( !plain && b_open );
      err_cl = is_cl !== ( !plain && !b_open );

      @( posedge clk_reactive );

      begin

         automatic bit checkable = !bad && !shadow_bad;

         automatic bit err_bad = bad !== shadow_bad;
         automatic bit err_lv = checkable && lev !== curr_depth;
         automatic bit err_await
           = checkable && lev && await !== shadow_await;
         string tr_txt;

         if ( err_op || err_cl || err_bad || err_lv || err_await )
           some_err++;

         num_tests++;
         errs_op += err_op;
         errs_cl += err_cl;
         errs_bad += err_bad;
         errs_lv += err_lv;
         errs_await += err_await;

         if ( !checkable ) bad_cyc++;

         tr_txt
           = $sformatf
             ("cyc %4d  s.c %2d.%2d  %1s  op %1h %2s  cl %1h %2s  bad %1h %2s  lev %2d %2d %2s  await '%1s%1s' %2s\n",
              cycle, s, c, in_char,
              is_op, oe[err_op],
              is_cl, oe[err_cl],
              bad, oe[err_bad],
              lev, curr_depth, oe[err_lv],
              await, shadow_await, oe[err_await] );

         trace_text.push_back(tr_txt);

         if ( some_err && groups_bad_count < show_groups_bad )
           while ( trace_text.size() ) $write( trace_text.pop_front() );
      end

      c++;
      @( negedge clock );
    end

    if ( !some_err && groups_good_count < show_groups_good )
           while ( trace_text.size() ) $write( trace_text.pop_front() );

    if ( some_err ) groups_bad_count++; else groups_good_count++;

  end
```

```
      done = 1;
   end

endmodule
// cadence translate_on
```

LSU EE 4755 Homework 6 Solution Due: 13 November 2017

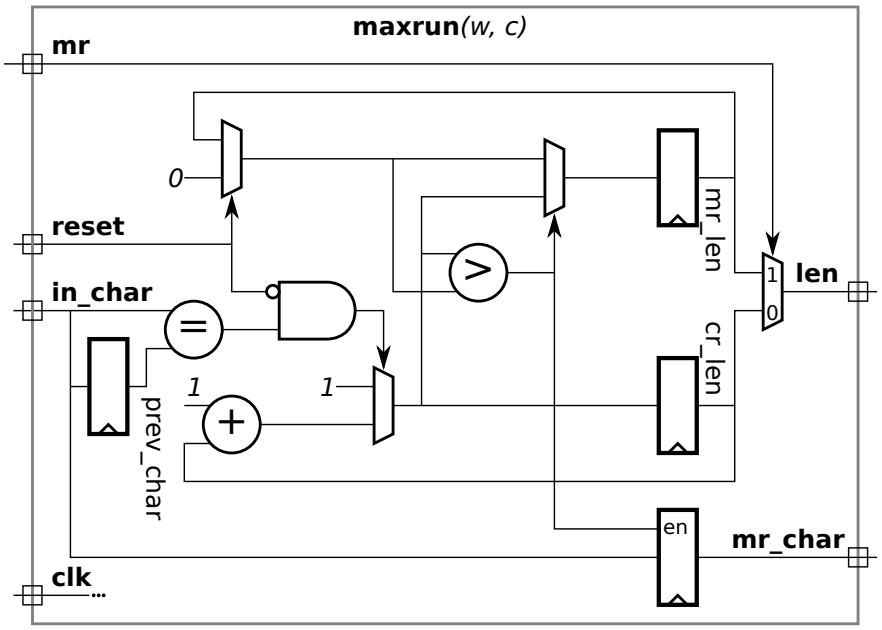
Problem 1: The solution to Homework 4, `http://www.ece.lsu.edu/koppel/v/2017/hw04-sol.v.html`, includes two modules, `maxrun` and `maxrun_opt`.

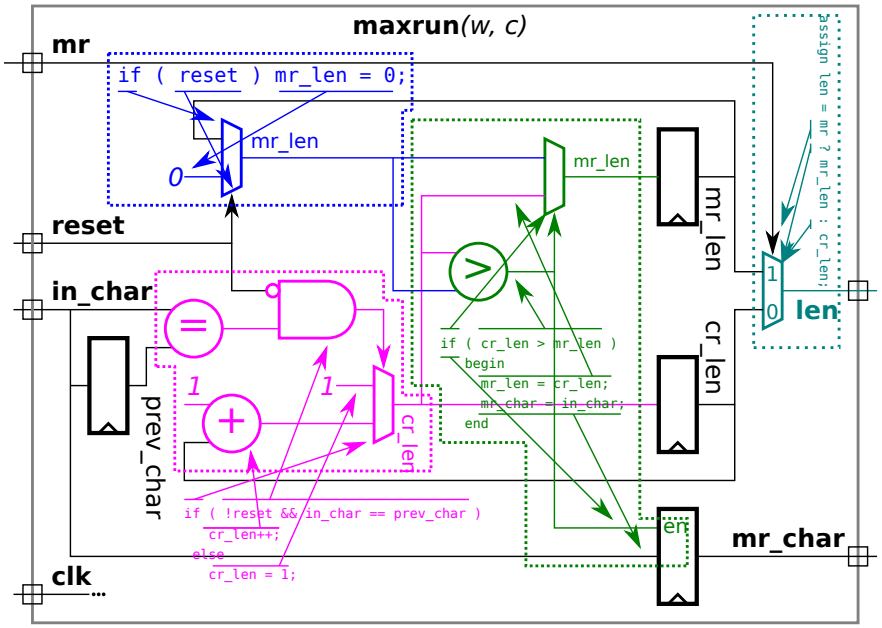
(*a*) Show the hardware inferred for `maxrun`. The Verilog code appears below.

```
module maxrun #( int w = 2, int c = 4 )
   ( output uwire [w-1:0] len,       output logic [c-1:0] mr_char,
     input uwire clk, reset, mr,     input uwire [c-1:0] in_char );
   logic [w-1:0] cr_len, mr_len;
   logic [c-1:0] prev_char;
   assign len = mr ? mr_len : cr_len;

   always_ff @( posedge clk ) begin

      if ( reset ) mr_len = 0;

      if ( !reset && in_char == prev_char )
        cr_len++;
      else
        cr_len = 1;

      if ( cr_len > mr_len )
        begin
           mr_len = cr_len;
           mr_char = in_char;
        end

      prev_char = in_char;
   end
endmodule
```

The solution appears below.

A common difficulty was properly accounting for order of assignments to `mr_len` and `cr_len`. The last assignment in the `always` block creates the value that is written to a register. The first illustration below shows the inferred hardware, the one below it shows the inferred hardware labeled with the Verilog code from which it was inferred.

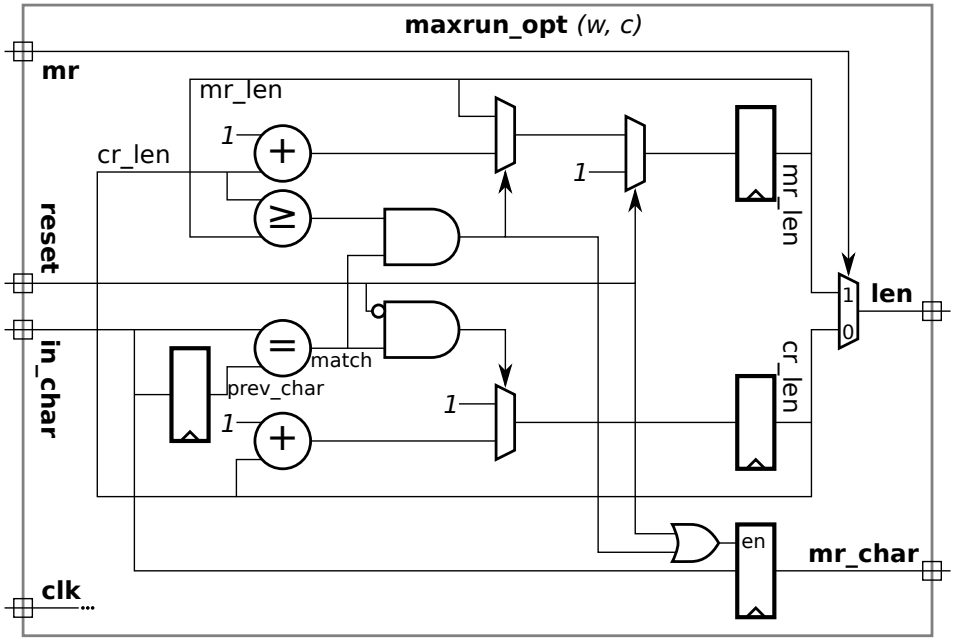




(*b*) Show the hardware inferred for `maxrun_opt`.

The solution appears below.

Note that there is no register for `match`. That is because it is not a live-out variable. That's obvious in this case because it is declared within the block.



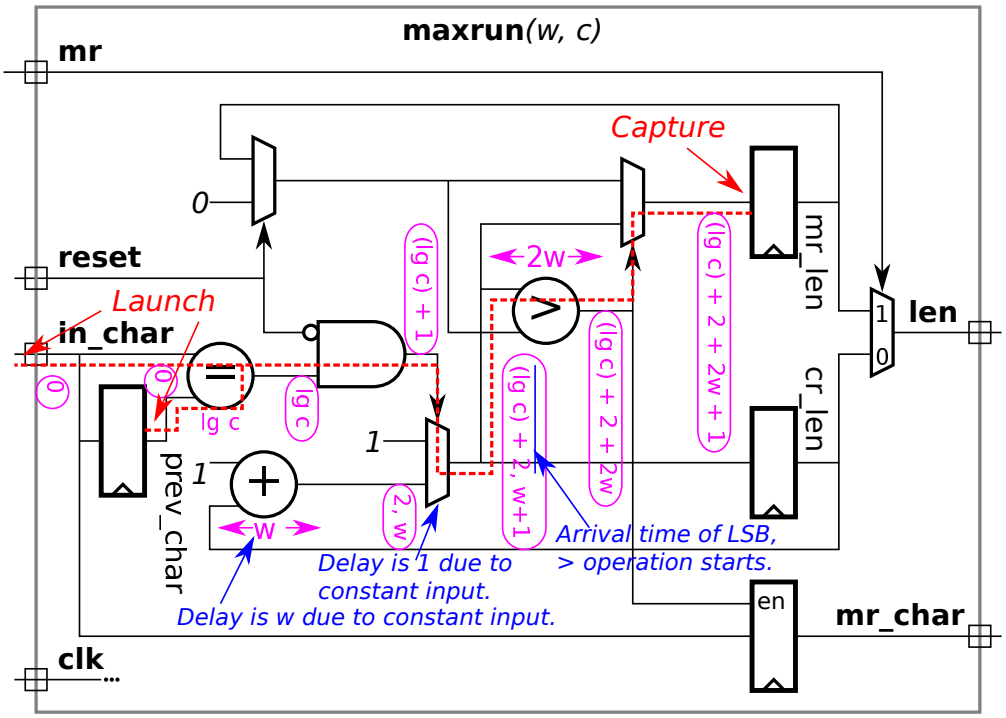
Problem 2: Compute the critical path for the `maxrun` and `maxrun_opt` modules using the simple model. The launch points (path starts) are at module inputs and register outputs, and the capture points (path ends) are at module outputs and register inputs. Note that with these definitions the critical path does not include the register itself. Show the critical path in terms of w, the number of bits in the `len` output and c, the number of bits in a character.

Short Answer: The critical path length is $(\lg c) + 2w + 3$ and its route is marked with a red dashed line in the illustration below. *Grading Note: In too many submissions the critical path was not marked on the diagram, instead relying on a prose description or just hints such as the modules the path passes through. Please show the path in the diagram.*

Explanation: The critical path starts at the `in_char` input and `prev_char` register output and follows the course shown. An interesting part of the critical path is the first mux on the path. The LSB of the lower data input arrives at $t = 2$ and the MSB arrives at $t = w$, which is later than the select signal, which arrives at $(\lg c) + 1$. Normally that would mean the lower data input is on the critical path. However, because the comparison unit can start when the LSB is ready the LSB arrival time determines criticality, and since $2 < (\lg c) + 1$ the select signal, not the data input, is on the critical path.

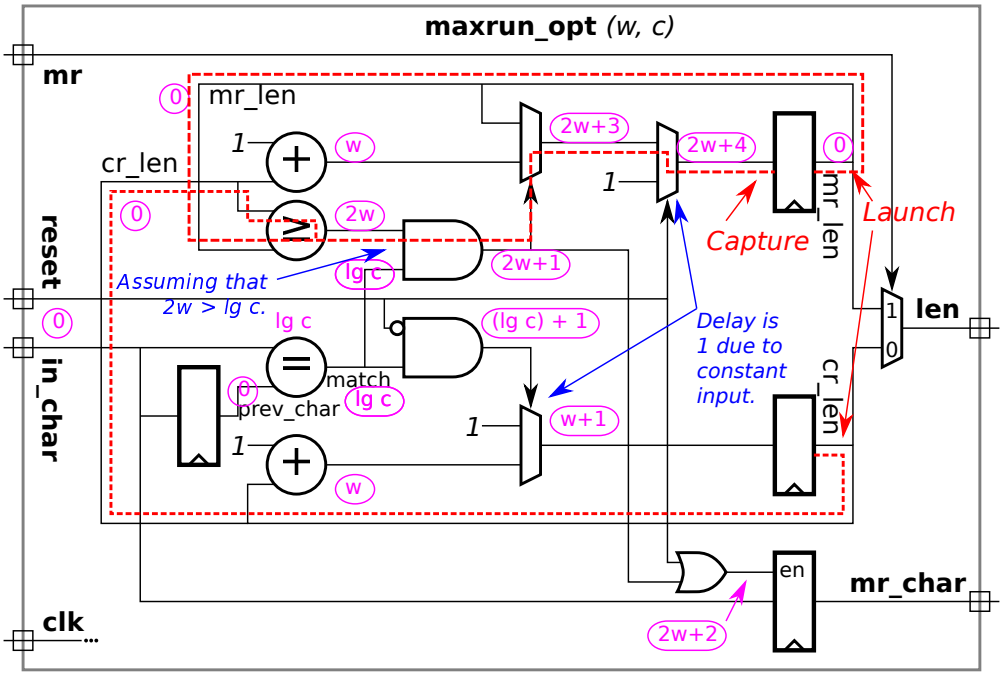
The `maxrun` module is slowed because the $>$ comparison must wait for the equality test.

Also note that multiplexors with constant inputs have a delay of 1 and that a w-bit ripple adder with a constant input has a delay of about w.

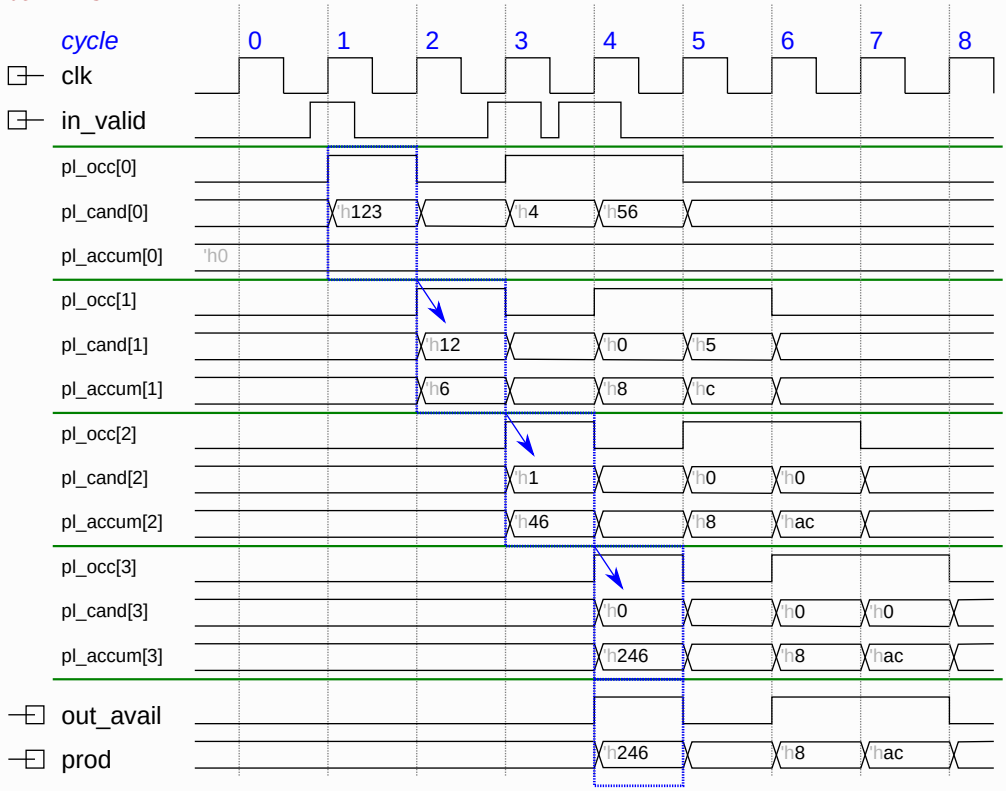


Short Answer: Assuming that $2w > \lg c$ the critical path length is $2w + 4$. The route of the path is shown with a red dashed line in the diagram.

Explanation: The critical path starts at the outputs of `mr_len` and `cr_len` and ends at the `mr_len` input. Unlike `maxrun`, the magnitude comparison and the equality test both start at $t = 0$, reducing the critical path.



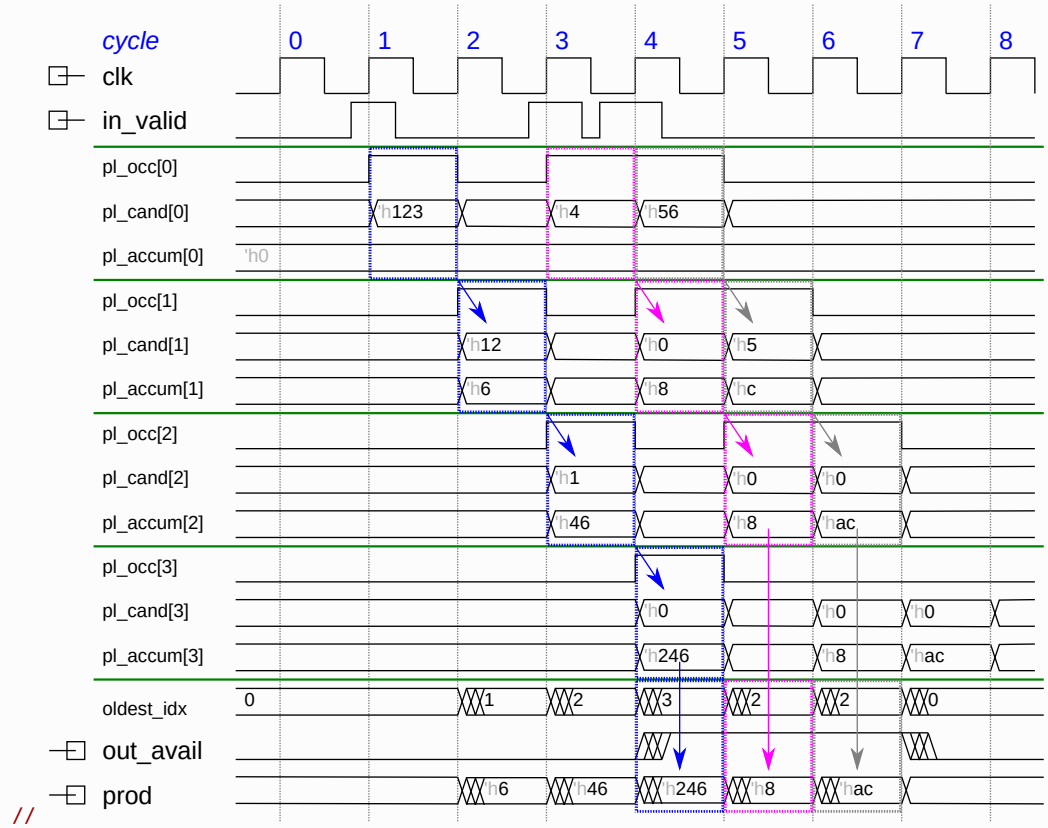
```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2017 Homework 7 -- SOLUTION
//

 /// Assignment http://www.ece.lsu.edu/koppel/v/2017/hw07.pdf


////////////////////////////////////////////////////////////////////////////////
///  Problem 1
//
 /// Complete so that mult_fast sets out_avail as described in the handout.
//
//     [✔] The module must be synthesizable.
//     [✔] Code must be reasonably efficient.
//     [✔] Do not change module parameters.
//     [✔] Do not change ports, EXCEPT changing between var and net kinds.
//     [✔] Don't assume that parameter values will match those used here.
//     [✔] USE DEBUGGING TOOLS LIKE SimVision.
//     [✔] Make sure that Avg cyc shown in testbench for Fast is lower
//         than Pipelined module of same degree (when all tests pass).

 /// SOLUTION - Problem 1a
//
// A new pipeline latch will be created, pl_occ, which will carry the
// value of in_valid through the pipeline. The pl_occ register for the
// last stage will connect to the output port out_avail.

// The timing diagram below is for a w=12, m=4 multiplier in which the
// multiplier is always set to 2 (not shown). Three different
// multiplicands arrive, 'h123, 'h4, and 'h56, their products, 2 *
// 'h123 = 'h246, 2 * 'h4 = 'h8, and 2 * 'h56 = 'hac, appear at the
// outputs three cycles later. The progress of 'h123 through the
// pipeline is highlighted in blue.
//
// The timing diagram does not show the plier and cand inputs. As
// stated above, the plier is always 2 (for the sample execution shown
// in the diagram). The value of input cand appearing at a positive
// edge can be seen in pl_cand[0] just after the positive edge. For
// example, at the positive edge between cycle 0 and 1 input cand must
// be 'h123.
```




```
//

 /// SOLUTION -- Problem 1a
//
module mult_fast_1a
  #( int w = 16,
     int m = 4 )
   ( output uwire [2*w-1:0] prod,
     output uwire out_avail,
     input uwire clk, in_valid,
```

```
      input uwire [w-1:0] plier, cand );

   localparam int nstages = ( w + m - 1 ) / m;

   logic [2*w-1:0] pl_accum[0:nstages];
   logic [w-1:0] pl_plier[0:nstages];
   logic [w-1:0] pl_cand[0:nstages];


   assign prod = pl_accum[nstages];

   /// SOLUTION -- Problem 1a
   //
   logic pl_occ[0:nstages];
   assign out_avail = pl_occ[nstages];
   //
   // Provide a pipeline latch for the in_valid signal to pass through
   // the pipeline and connect the last stage's latch to the out_avail
   // port.

   always_ff @( posedge clk ) begin

      /// SOLUTION -- Problem 1a
      //
      pl_occ[0] = in_valid;
      //
      // Connect in_valid port to the first stage.

      pl_accum[0] = 0;
      pl_plier[0] = plier;
      pl_cand[0] = cand;

      for ( int stage=0; stage<nstages; stage++ ) begin

         pl_accum[stage+1] <=
           pl_accum[stage] +
             ( pl_plier[stage] * pl_cand[stage][m-1:0] << stage*m );

         pl_cand[stage+1] <= pl_cand[stage] >> m;
         pl_plier[stage+1] <= pl_plier[stage];

         /// SOLUTION -- Problem 1a
         //
         pl_occ[stage+1] <= pl_occ[stage];
         //
         // Pass the in_valid signal through the pipeline.

      end

   end

endmodule

 /// SOLUTION -- Problem 1b (and also 1a)
//
// The fast multiplier is supposed to provide a shortcut connection
// from each stage to the multiplier output, prod. Stage x can use the
// shortcut connection if the multiplication that the stage is
// carrying is complete and if no higher-numbered stages are occupied.
// A multiplication (meaning the result of multiplying a plier and
// cand) must appear at the output exactly once, and the arriving
// multiplications must appear at the outputs in the same order in
// which they arrived.

 /// The Plan
//
//  - Find the highest-numbered occupied stage. Call it oldest_idx.
//  - Connect the result at that stage, pl_accum[oldest_idx], to prod.
//  - Set out_avail to true if stage oldest_idx is finished.
//  - Set pl_occ[oldest_idx] to zero if out_avail is true, to avoid duplicates.
//
// For the discussion below refer to module mult_fast_1b and to the
// timing diagram below. The timing diagram is for a module
// instantiated with w=12, m=4, and for which the multiplier is always
// 2. The arriving values in the timing diagram below are the same as
// the diagram appearing in the solution to Problem 1a.

 /// Computing oldest_idx
//
// Combinational logic will be added that computes oldest_idx, the
// highest-numbered occupied stage. (Stage x is occupied if pl_occ[x]
// is true.) If none of the stages are occupied oldest_idx is set to
// zero.
//
// See the always_comb block in mult_fast_1b below.

 /// Connect Stage's Result to Prod
```

```
//
// Output prod is set to pl_accum[oldest_idx], see the assign in
// mult_fast_1b. The connection is made whether or not stage
// oldest_idx is finished or contains a valid value.

// In the timing diagram notice that at cycles 2 and 3 prod holds
// incomplete multiplications. That's fine because out_avail is zero.
// Setting prod to something like zero at cycles like 2 and 3 would
// require extra hardware and provide no benefit (based on the problem
// statement in the homework handout).

 /// Set out_avail
//
// Output out_avail is set to 1 if stage oldest_idx is occupied and if
// the multiplicand at that stage is zero, meaning that the
// multiplication is complete. Whether it is occupied can be
// determined by examining pl_occ[oldest_idx], whether it is finished
// can be determined by examining pl_cand[oldest_idx]. See the "assign
// out_avail" in mult_fast_1b.

 /// Set pl_occ[oldest_idx] When Done
//
// Since the result of a multiplication cannot appear at the output
// more than once pl_occ[x] must be set to zero at the end of cycle c
// if x was chosen in cycle c. (If this were not done the same
// multiplication would appear at the outputs again in the next clock
// cycle [c+1], when it is in stage x+1.) For example, stage 2 is
// chosen in cycle 5, its product 'h8 appears at the output. In cycle
// 6 the calculation is in stage 3, but now its value of pl_occ is
// zero and so it won't be chosen a second time.
```




```
//
module mult_fast_1b
  #( int w = 16,
     int m = 4 )
   ( output uwire [2*w-1:0] prod,
     output uwire out_avail,
     input uwire clk, in_valid,
     input uwire [w-1:0] plier, cand );

   localparam int nstages = ( w + m - 1 ) / m;

   logic [2*w-1:0] pl_accum[0:nstages];
   logic [w-1:0] pl_plier[0:nstages], pl_cand[0:nstages];
   logic pl_occ[0:nstages];

   /// SOLUTION -- Problem 1b
   //
   //  Determine the idx of the last (highest-numbered) occupied
   //  stage, in which resides the oldest multiplication in the
```

```
   //  pipeline.
   //
   logic [$clog2(nstages):0] oldest_idx;
   //
   always_comb begin
      oldest_idx = 0;
      for ( int i=1; i<=nstages; i++ ) if ( pl_occ[i] ) oldest_idx = i;
   end

   /// SOLUTION -- Problem 1b
   //
   //  Connect the last occupied stage to the output ..
   //
   assign prod = pl_accum[oldest_idx];
   //
   //  .. and set out_avail to true if that stage is occupied and finished.
   //
   assign out_avail = pl_occ[oldest_idx] && pl_cand[oldest_idx] == 0;


   always_ff @( posedge clk ) begin

      pl_occ[0] = in_valid;

      pl_accum[0] = 0;
      pl_plier[0] = plier;
      pl_cand[0] = cand;

      for ( int stage=0; stage<nstages; stage++ ) begin

         pl_accum[stage+1] <=
           pl_accum[stage] +
             ( pl_plier[stage] * pl_cand[stage][m-1:0] << stage*m );

         pl_cand[stage+1] <= pl_cand[stage][w-1:m];
         pl_plier[stage+1] <= pl_plier[stage];

         /// SOLUTION -- Problem 1b
         //
         //  Pass 0 to next stage if this stage is providing the
         //  result, otherwise pass this stage's value of occupied to
         //  the next stage.
         //
         pl_occ[stage+1] <=
           oldest_idx == stage && out_avail ? 0 : pl_occ[stage];

      end

   end

endmodule




module mult_behav_1
  #(int w = 16)
   (output logic [2*w-1:0] prod, input logic [w-1:0] plier, cand);

   assign prod = plier * cand;
endmodule

 /// :Example: Basic Pipelined Multiplier -- mult_pipe
//
//   Computes m partial products per stage.
//

module mult_pipe #( int w = 16, int m = 4 )
   ( output logic [2*w-1:0] prod,
     input logic [w-1:0] plier,
     input logic [w-1:0] cand,
     input clk);

   localparam int nstages = ( w + m - 1 ) / m;

   // Note: pl is for pipeline latch.
   logic [2*w-1:0] pl_accum[0:nstages];
   logic [w-1:0] pl_plier[0:nstages];
   logic [w-1:0] pl_cand[0:nstages];

   always_ff @( posedge clk ) begin

      pl_accum[0] = 0;
      pl_plier[0] = plier;
      pl_cand[0] = cand;

      for ( int stage=0; stage<nstages; stage++ ) begin
```

```
         logic [w-1:0] cand_next;
         cand_next = pl_cand[stage][w-1:m];

         pl_accum[stage+1] <=
           pl_accum[stage] +
             ( pl_plier[stage] * pl_cand[stage][m-1:0] << stage*m );

         pl_cand[stage+1] <= cand_next;
         pl_plier[stage+1] <= pl_plier[stage];


      end

   end

   assign prod = pl_accum[nstages];

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Testbench Code

// cadence translate_off

program reactivate
   (output uwire clk_reactive, output int cycle_reactive,
    input uwire clk, input int cycle);
   assign clk_reactive = clk;
   assign cycle_reactive = cycle;
endprogram

module testbench;

   localparam int w = 16;
   localparam int num_tests = 400;
   localparam int NUM_MULT = 20;
   localparam int err_limit = 7;

   bit use_others;
   logic [w-1:0] plier, cand;
   logic [w-1:0] plierp[NUM_MULT], candp[NUM_MULT];
   logic [2*w-1:0] prod[NUM_MULT];
   uwire availn[NUM_MULT];
   logic avail[NUM_MULT];
   logic in_valid[NUM_MULT];

   typedef struct { int tidx; int cycle_start; } Test_Vector;

   typedef struct { int idx;
                    int err_count = 0;
                    int err_timing = 0;
                    Test_Vector tests_active[$];
                    bit all_tests_started = 0;
                    bit seq = 0; bit pipe = 0;
                    bit bpipe = 0;
                    int deg = 1;
                    int ncompleted = 0;
                    int cyc_tot = 0;
                    int latency = 0;
                    } Info;
   Info pi[string];

   localparam int cycle_limit = num_tests * w * 4;
   int cycle;
   bit done;
   logic clock;

   logic clk_reactive;
   int cycle_reactive;
   reactivate ra(clk_reactive,cycle_reactive,clock,cycle);


   initial begin
      clock = 0;
      cycle = 0;

      fork
         forever #10 cycle += clock++;
         wait( done );
         wait( cycle >= cycle_limit )
           $write("*** Cycle limit exceeded, ending.\n");
      join_any;

      $finish();
```

```
end

task pi_seq(input int idx, input string name, input int deg);
   automatic string m = $sformatf("%s Deg %0d", name, deg);
   pi[m].deg = deg;
   pi[m].idx = idx; pi[m].seq = 1; pi[m].bpipe = 0;
endtask

task pi_pipe(input int idx, input string name, input int deg);
   automatic string m = $sformatf("%s Deg %0d", name, deg);
   pi[m].deg = deg;
   pi[m].idx = idx; pi[m].seq = 1; pi[m].pipe = 1; pi[m].bpipe = 0;
endtask
task pi_bpipe(input int idx, input string name, input int deg);
   automatic string m = $sformatf("%s Deg %0d", name, deg);
   pi[m].deg = deg;
   pi[m].idx = idx; pi[m].seq = 1; pi[m].pipe = 1; pi[m].bpipe = 1;
endtask

mult_behav_1 #(w) mb1(prod[0], plierp[0], candp[0]);
initial pi["Behavioral"].idx = 0;

mult_pipe    #(w,4) ms2(prod[2], plierp[2], candp[2], clock);
initial pi_pipe(2,"Pipelined",ms2.m);

mult_pipe    #(w,2) ms3(prod[3], plierp[3], candp[3], clock);
initial pi_pipe(3,"Pipelined",ms3.m);

mult_fast_1a    #(w,4) ms7(prod[7], availn[7], clock,
                           in_valid[7], plierp[7], candp[7]);
initial pi_bpipe(7,"Fast 1a",ms7.m);

mult_fast_1a    #(w,2) ms8(prod[8], availn[8], clock,
                           in_valid[8], plierp[8], candp[8]);
initial pi_bpipe(8,"Fast 1a",ms8.m);

mult_fast_1a    #(w,1) ms9(prod[9], availn[9], clock,
                           in_valid[9], plierp[9], candp[9]);
initial pi_bpipe(9,"Fast 1a",ms9.m);

mult_fast_1b    #(w,4) ms17(prod[17], availn[17], clock,
                           in_valid[17], plierp[17], candp[17]);
initial pi_bpipe(17,"Fast 1b",ms17.m);

mult_fast_1b    #(w,2) ms16(prod[16], availn[16], clock,
                           in_valid[16], plierp[16], candp[16]);
initial pi_bpipe(16,"Fast 1b",ms16.m);

mult_fast_1b    #(w,1) ms15(prod[15], availn[15], clock,
                           in_valid[15], plierp[15], candp[15]);
initial pi_bpipe(15,"Fast 1b",ms15.m);

always @*
  foreach ( availn[i] ) if ( availn[i] !== 1'bz ) avail[i] = availn[i];

// Array of multiplier/multiplicand values to try out.
// After these values are used a random number generator will be used.
//
int tests[$] = {1,1, 1,2, 1,3, 1,4, 1,5,  1,32,  32, 1};

initial begin

   automatic int awaiting = pi.size();

   logic [w-1:0] pliers[num_tests], cands[num_tests];

   done = 0;

   foreach ( pi[mut] ) begin
      automatic int midx = pi[mut].idx;
      automatic int steps = ( w + pi[mut].deg - 1 ) / pi[mut].deg;
      automatic int latency =
        !pi[mut].seq ? 1 : !pi[mut].pipe ? 2 * steps : steps;
      pi[mut].latency = latency;
      if ( pi[mut].bpipe == 0 ) begin
         avail[midx] = 1;
      end
      in_valid[midx] = 0;
   end

   for ( int i=0; i<num_tests; i++ ) begin

      automatic int num_bits_c = {$random()}%w + 1;
      automatic logic [w-1:0] mask_c = ( (w+1)'(1) << num_bits_c ) - 1;
      automatic int num_bits_p = {$random()}%w + 1;
      automatic logic [w-1:0] mask_p = ( (w+1)'(1) << num_bits_p ) - 1;
```

```
      pliers[i] = tests.size() ? tests.pop_front() : {$random()}&mask_p;
      cands[i] = tests.size() ? tests.pop_front() : {$random()}&mask_c;

   end

   fork forever @( negedge clk_reactive ) foreach ( pi[mut] ) begin
      automatic int midx = pi[mut].idx;
      if ( !in_valid[midx] && pi[mut].pipe ) begin
         plierp[midx] = cycle;
         candp[midx] = 1;
      end
   end join_none;

   repeat ( 2 * w ) @( negedge clock );

   foreach ( pi[mutii] ) begin
      automatic string muti = mutii;

      fork begin
         automatic string mut = muti;
         automatic int midx = pi[mut].idx;
         for ( int i=0; i<num_tests; i++ ) begin
            automatic int gap_cyc =
              ( {$random} % 2 ) ? {$random} % ( w + 2 ) : 0;
            automatic Test_Vector tv;
            repeat ( gap_cyc ) @( negedge clock );
            plierp[midx] = pliers[i];
            candp[midx] = cands[i];
            in_valid[midx] = 1;
            tv.tidx = i;
            tv.cycle_start = cycle;
            pi[mut].tests_active.push_back( tv );
            @( negedge clock );
            in_valid[midx] = 0;
         end
         pi[mut].all_tests_started = 1;
      end join_none;

      fork begin
         automatic string mut = muti;
         automatic int midx = pi[mut].idx;
         while ( 1 ) begin
            @( negedge clock );
            while ( pi[mut].tests_active.size() == 0
                    && !pi[mut].all_tests_started )
              @( negedge clock );
            if ( pi[mut].tests_active.size() == 0 ) break;
            begin
               automatic Test_Vector tv = pi[mut].tests_active.pop_front();
               automatic int i = tv.tidx;
               automatic logic [2*w-1:0] shadow_prod = pliers[i] * cands[i];
               automatic int eta = tv.cycle_start + pi[mut].latency;
               automatic bit timing_err = 0;
               automatic int delta_t;
               if ( pi[mut].bpipe ) begin
                  while ( !avail[midx] && cycle < eta ) @( negedge clock );
                  if ( !avail[midx] || cycle > eta ) begin
                     timing_err = 1;
                     if ( pi[mut].err_timing++ < err_limit )
                       $write("At cyc %4d (eta %0d) avail not set for %s (idx %0d) after %0d cycles for 0x%0h*0x%0h.\n",
                              cycle, eta, mut, midx, cycle - tv.cycle_start,
                              pliers[i], cands[i]);
                  end
               end else begin
                  wait ( cycle >= eta );
               end
               delta_t = cycle - tv.cycle_start;
               if ( !timing_err ) begin
                  pi[mut].ncompleted++;
                  pi[mut].cyc_tot += delta_t;
               end
               if ( !timing_err && shadow_prod !== prod[midx] ) begin
                  pi[mut].err_count++;
                  if ( pi[mut].err_count < err_limit ) begin
                     $write
                       ("%-15s test %5d  cyc %0d+%0d (%0d) wrong: 0x%0h * 0x%0h:  0x%0h != 0x%0h (correct)\n",
                        mut, i, tv.cycle_start, delta_t, pi[mut].latency,
                        pliers[i], cands[i],
                        prod[midx], shadow_prod);
                  end
               end
            end
         end
         awaiting--;
      end join_none;

   end
```

```
      wait( awaiting == 0 || cycle > cycle_limit );

      $write("At cycle %0d.  Error types:  couldn't test / wrong result / timing\n",cycle);

      foreach ( pi[ mut ] )
        $write("For %-18s ran %4d tests, %4d/%4d/%4d errors found. Avg cyc %.1f\n",
               mut, num_tests,
               num_tests - pi[mut].ncompleted,
               pi[mut].err_count, pi[mut].err_timing,
               pi[mut].seq ? real'(pi[mut].cyc_tot) / pi[mut].ncompleted : 1);

      done = 1;

      $finish(2);

   end

endmodule

// cadence translate_on
```

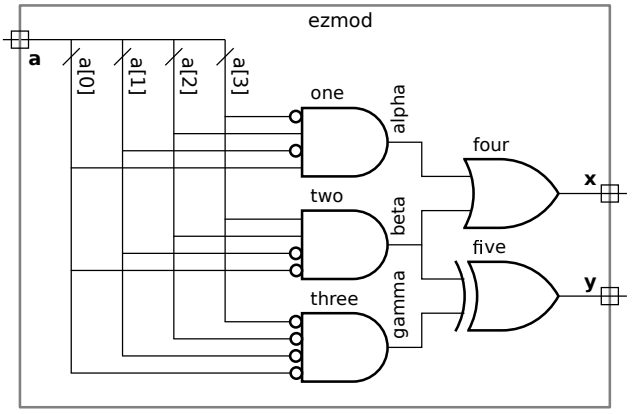
22 Fall 2016 Solutions

LSU EE 4755 Homework 1 Solution Due: 9 September 2016

The questions below can be answered without using EDA software, paper and pencil will suffice. Please turn in the solution on paper. Homework 2 will require the use of Verilog implementations. Nevertheless, runnable SystemVerilog code for this assignment can be found at `https://www.ece.lsu.edu/koppel/v/2016/hw01.v` *(plain Verilog) and* `https://www.ece.lsu.edu/koppel/v/2016/hw01.v.html` *(syntax-highlighted HTML).*

Those who are rusty about the correspondence between Verilog code and hardware might want to look at the solution to EE 3755 Fall 2013 Homework 1, at `http://www.ece.lsu.edu/ee3755/2013f/hw01_sol.pdf`.

Problem 1: Show a Verilog explicit structural description of the module illustrated below. In this assignment it is okay to use primitives (`and, not,`...), but don't get in the habit of using them.



- Base the names of ports, wires, and instances on labels in the illustration.
- Of course, use only primitives and wires. See Table 28-1 of IEEE Std 1800-2012 for a list of gates.

Solution appears below. In order to be explicitly structural NOT gates were instantiated to provide the inverted inputs for the AND gates. In real life, there would be no disadvantage using `!a[2]` in place of `na2`. (That may not be 100% true, because working for a company with super-strict HDL style rules is a real-life situation.)

```
module ezmod( output uwire x, y,   input uwire [3:0] a ); // SOLUTION
   uwire   na0, na1, na2, na3;
   not n0(na0,a[0]);
   not n1(na1,a[1]);
   not n2(na2,a[2]);
   not n3(na3,a[3]);

   uwire   alpha, beta, gamma;
   and one(alpha,   na3,  a[2], na1,  a[0] );
   and two(beta,    a[3], a[2], na1,  na0  );
   and three(gamma, na3,  na2,  na1,  na0  );

   or four(x, alpha, beta );
   xor five(y, beta, gamma );
endmodule
```

Problem 2: Answer the following questions about Verilog primitives as defined in IEEE Std 1800-2012. (See Chapter 28.) Indicate the exact section number where the answer is found.

(*a*) The standard provides a `not` primitive and a `nor` primitive, among others. One can easily argue that a 1-input `nor` gate is the same as a `not` gate. Does the standard actually allow Verilog code to instantiate a 1-input `nor` gate?

Yes, see Section 28.4.

Grading Note: It is not correct to answer "table 28-1 because it is shown as an n-input gate", because the table does not explicitly state that $n == 1$ is acceptable for a `nor` gate.

(*b*) Based on the standard, is there anything that can be done with a `not` primitive that can't be done with a 1-input `nor` primitive? (Don't try to answer this too deeply, just show an instantiation.)

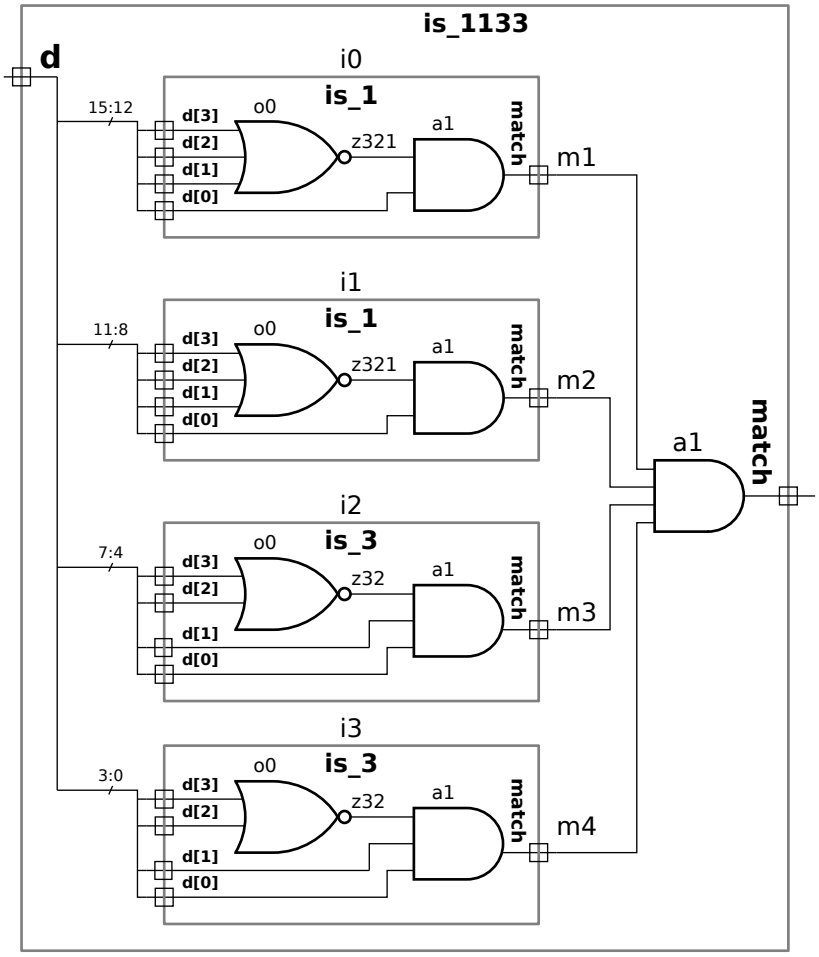
Yes, a `not` primitive can have more than one output. The outputs all have the same value under unstressed circumstances. Multiple-output `not` gates will not be used for designs in this class.

Problem 3: Output `match` of module `is_1133`, shown below, is `1` iff its input `d` (digits) is 1133 in BCD (which has the same representation as 1133_16). The module instantiates BCD digit detection modules `is_1` and `is_3`.

```
module is_1( output uwire match, input uwire [3:0] d );
   uwire   z321;
   nor o0(z321,d[3],d[2],d[1]);
   and a1(match,z321,d[0]);
endmodule

module is_3( output uwire match, input uwire [3:0] d );
   uwire z32;
   nor o0(z32,d[3],d[2]);
   and a1(match,z32,d[1],d[0]);
endmodule

module is_1133( output uwire match, input uwire [15:0] d );
   uwire          m1, m2, m3, m4;

   and a1(match, m1, m2, m3, m4);

   is_1 i0(m1, d[15:12]);
   is_1 i1(m2, d[11:8]);
   is_3 i2(m3, d[7:4]);
   is_3 i3(m4, d[3:0]);
endmodule
```

(*a*) Draw a diagram of `is_1133` based on the explicit structural description above. Show the insides of the `is_1` and `is_3` modules. Label the diagram using the same wire and instance names used in the Verilog descriptions.

Solution appears to the right.



(*a*) Design a module `is_1133_is` that does the same thing as `is_1133`, but that uses implicit structural code. The correct solution requires adding only one short line to the shell shown below. Don't forget that the value in `d` is in BCD. *Note: The word* short *was added after the original assignment.*

Solution appears below. The comparison operator checks for the correct value. We need to compare `d` to the BCD representation of 1133. Verilog does not have literal format just for BCD, as it does for binary, octal, decimal, and hexadecimal. But it doesn't need one because the BCD representation of 1133 is the same as the binary representation of 1133_16, which in Verilog is $16'h1133$. That means that the `is_1333` module (either version) has an output that's one iff the input is the BCD representation of 1333 or the unsigned binary representation of 4403 (because $4403\_10 = 1133\_16$).

```
// SOLUTION
module is_1133_is( output uwire match, input uwire [15:0] d );
   assign        match = d == 16'h1133;
endmodule
```

Problem 4: When completed the output of module `is_1235` is 1 iff the input is 1235 in BCD.

```
module is_1235( output uwire match, input uwire [15:0] d );

endmodule
```

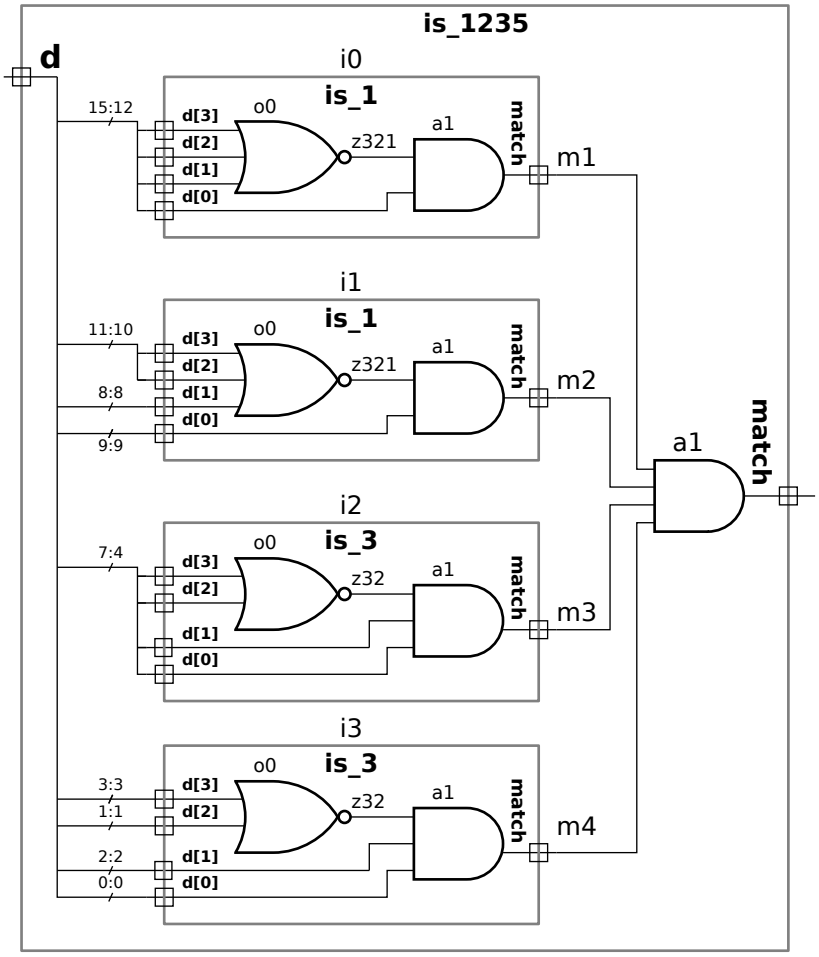
(*a*) Complete the module. The module must be explicitly structural except for the use of the concatenation operator (see Section 11.4.12). The module **must** use `is_1` and `is_3` to detect the digits. Do not assume or design an `is_2` or `is_5` and don't put in logic to detect those digits.

Solution appears below. The `is_1` module is used to detect a 2 by swapping the two least-significant bits. (The same method can be used to detect a 4 or an 8.) Similarly, the `is_3` is used to detect a 5 by swapping the two middle digits. (The same method can be used to detect a 6 or a 9.)

```
// SOLUTION
module is_1235( output uwire match, input uwire [15:0] d );
   uwire  m1, m2, m3, m4;
   is_1 i0(m1, d[15:12]);
   is_1 i1(m2, {d[11:10],d[8],d[9]});   // Actually detect 2.
   is_3 i2(m3, d[7:4]);
   is_3 i3(m4, {d[3],d[1],d[2],d[0]} ); // Actually detect 5.
   and a1(match, m1, m2, m3, m4);
endmodule
```

(*b*) Draw a diagram of the completed module, which should be very similar to the diagram from the previous problem.

Solution appears to the right.



```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2016 Homework 2 -- SOLUTION
//

 /// Assignment http://www.ece.lsu.edu/koppel/v/2016/hw02.pdf


`default_nettype none

////////////////////////////////////////////////////////////////////////////////
///  Problem 1
//
 /// Modify aa_digit_val so that it works for any radix, not just 10.
//
//     [✔] The code must be synthesizable.
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Can use behavioral or implicit structural code.

module aa_decimal_digit_val
  ( output uwire [3:0] val,
    output uwire is_dig,
    input uwire [7:0] char );

   // Do not edit this module.

   assign       is_dig = char >= "0" && char <= "9";
   assign       val = is_dig ? char - "0" : 0;

endmodule

module aa_digit_val
  #( int radix = 10 )
   ( output uwire [3:0] val,
     output uwire is_dig,
     input uwire [7:0] char );

   /// SOLUTION

   // Check whether char is in range 0-9 or a-f, regardless of radix.
   //
   uwire  is_dig_09 = char >= "0" && char <= "9";
   uwire  is_dig_af = char >= "a" && char <= "f";

   // Convert char to binary, assuming that it is hexadecimal.
   //
   uwire [3:0] val_raw = is_dig_09 ? char - "0" : char - "a" + 10;

   // Determine whether char is a valid digit in radix radix.
   //
   assign     is_dig = ( is_dig_09 || is_dig_af ) && val_raw < radix;

   assign     val = is_dig ? val_raw : 0;

endmodule


////////////////////////////////////////////////////////////////////////////////
///  Problem 2
//
 /// Modify aa_full_adder so that it adds two radix-RADIX ASCII-encoded digits.
//
//     [✔] The code must be synthesizable.
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Can use behavioral or implicit structural code.
```

```
module aa_full_adder
  #( int radix = 10 )
   ( output uwire [7:0] sum,
     output uwire carry_out,
     output uwire is_dig_out,
     input uwire [7:0] a, b,
     input uwire carry_in,
     input uwire is_dig_in);

   /// SOLUTION

   // Instantiate two aa_digit_val modules, connecting one to each
   // input digit. These will determine whether each character input
   // is a valid digit and if so provide the binary value of the
   // digit.
   //
   uwire [3:0] val_a, val_b;
   uwire       is_dig_a, is_dig_b;
   aa_digit_val #(radix) dva(val_a, is_dig_a, a);
   aa_digit_val #(radix) dvb(val_b, is_dig_b, b);

   // Compute the sum of carry_in and the binary versions of a and
   // b. Note that the sum may contain a carry, and so it can not be
   // assigned to the module output.
   //
   uwire [4:0] sum_val = carry_in + val_a + val_b;

   // Determine whether there is a carry out.
   //
   assign     carry_out = sum_val >= radix;

   // Determine the sum, in binary, with the carry removed.
   //
   uwire [3:0] sum_dig_val = carry_out ? sum_val - radix : sum_val;

   // Convert the sum to ASCII or to a blank if we don't have a valid digit.
   //
   assign sum = !is_dig_out ? " " :
          sum_dig_val < 10 ? "0" + sum_dig_val : "a" + sum_dig_val - 10;


   // If the value of is_dig_out, below, is true then output sum will
   // be set to a digit of the sum. Otherwise sum should be set to a
   // blank. The value of is_dig_out will be false when we are past
   // the last digit of both a and b, and we don't have a carry out
   // from the previous digit.
   //
   assign is_dig_out = is_dig_in && ( carry_in || is_dig_a || is_dig_b );


endmodule


module aa_width2
  #( int radix = 10 )
   ( output uwire [1:0][7:0] sum,
     output uwire c_out,
     output uwire is_dig_out,
     input uwire [1:0][7:0] a, b,
     input uwire c_in,
     input uwire is_dig_in);

   uwire   co0, id_0;

   aa_full_adder #(radix) fa1(sum[0],co0,id_0,a[0],b[0],c_in,is_dig_in);
   aa_full_adder #(radix) fa2(sum[1],c_out,is_dig_out,a[1],b[1],co0,id_0);

endmodule
```

```
module reference_adder
  #( int radix = 10,
     int digits = 2,
     int width = $clog2( radix ** digits ) )
   ( output logic [width-1:0] sum,
     output logic carry_out,
     input uwire [width-1:0] a, b,
     input uwire carry_in );

   always_comb { carry_out, sum } = 0 + carry_in + a + b;

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Testbench Code
//
//  The code below instantiates some of the modules above,
//  provides test inputs, and verifies the outputs.
//
//  The testbench may be modified to facilitate your solution. Of
//  course, the removal of tests which your module fails is not a
//  method of fixing a broken module. (One might modify the testbench
//  so that the first tests it performs are thoe which make it easier
//  to determine what the problem is, for example, test inputs that
//  are all 0's or all 1's.)


// cadence translate_off


// Convert integer A into a radix RADIX ASCII representation.
function automatic string radtos(int unsigned a, int radix);
      begin
         radtos = "";
         while ( 1 )
           begin
              automatic int dig = a % radix;
              if ( radtos.len() > 0 && a == 0 ) break;
              a = a / radix;
              radtos = { dig < 10 ? "0" + dig : "a" + dig - 10, radtos };
           end
      end
endfunction

module aa_test;

   logic start_done[32];

   for ( genvar radix = 2; radix <= 16; radix++ )
     aa_test_digit_val #(radix) aa(start_done[radix],start_done[radix-1]);

   for ( genvar radix = 2; radix <= 16; radix++ )
     aa_test_width2 #(radix) aa(start_done[15+radix],start_done[15+radix-1]);

   initial begin

      start_done[1] = 1;

   end

endmodule

module aa_test_digit_val #( int radix = 10 )
   ( output logic done, input uwire start );

   localparam int err_limit = 10;
```

```
   logic [7:0] a;
   uwire [3:0]  dval;
   uwire        is_d;

   aa_digit_val #(radix) dv1(dval,is_d,a);

   int digit_vals[256];

   int num_errs;

   initial begin
      num_errs = 0;

      wait ( start == 1 );

      digit_vals = { 256 { -1 } };
      for ( int i=0; i<radix; i++ )
        digit_vals[ i < 10 ? "0" + i : "a" + i - 10 ] = i;

      for ( int i=0; i<256; i++ )
        begin
           automatic bit is_d_shadow = digit_vals[i] >= 0 ;
           #1;
           a = i;
           #1;
           if ( is_d != is_d_shadow ) begin
              $write
                ("Error in aa_digit_val for char %c (%0d) radix %0d, is_d %0d != %0d (correct)\n",
                 i, i, radix, is_d, is_d_shadow );
              num_errs++;
              if ( num_errs > err_limit ) $finish(2);
              continue;
           end
           if ( is_d_shadow && dval != digit_vals[i] ) begin
              $write
                ("Error in aa_digit_val for char %c (%0d) radix %0d: val %0d != %0d (correct)\n",
                 i, i, radix, dval, digit_vals[i]);
              num_errs++;
              if ( num_errs > err_limit ) $finish(2);
              continue;
           end

        end

      done = 1;

   end

endmodule

module aa_test_width2 #( int radix = 10 )
   ( output logic done, input uwire start );

   localparam int err_limit = 10;
   localparam int max_digits = 2;
   localparam int max_dno = max_digits - 1;
   localparam int num_tests = 100;

   uwire [max_dno:0][7:0] sum;
   logic [max_dno:0][7:0] a, b, shadow_sum;
   uwire        co;
   logic       fo, ci, fi;

   aa_width2 #(radix) fa1(sum[1:0],co,fo,a[1:0],b[1:0],ci,fi);

   int unsigned aval, bval, ssum_val;

   int num_errs;
```

```
   initial begin
      num_errs = 0;

      wait ( start == 1 );

      for ( int i=0; i<num_tests; i++ )
        begin
           automatic int width = max_digits;
           aval = {$random()} >> 1;
           bval = {$random()} >> 1;
           ssum_val = aval + bval;
           a = radtos(aval,radix);
           b = radtos(bval,radix);
           shadow_sum = radtos(ssum_val,radix);
           ci = 0;
           fi = 1;
           #1;

           if ( sum !== shadow_sum ) begin

              $write("Error %s + %s != %s (%s correct).\n",
                     a,b,sum,shadow_sum);
              num_errs++;
              if ( num_errs > err_limit ) $finish(2);

           end

           #1;

        end

      done = 1;

   end

endmodule


// cadence translate_on
```

LSU EE 4755 Homework 3 Solution Due: 28 September 2016

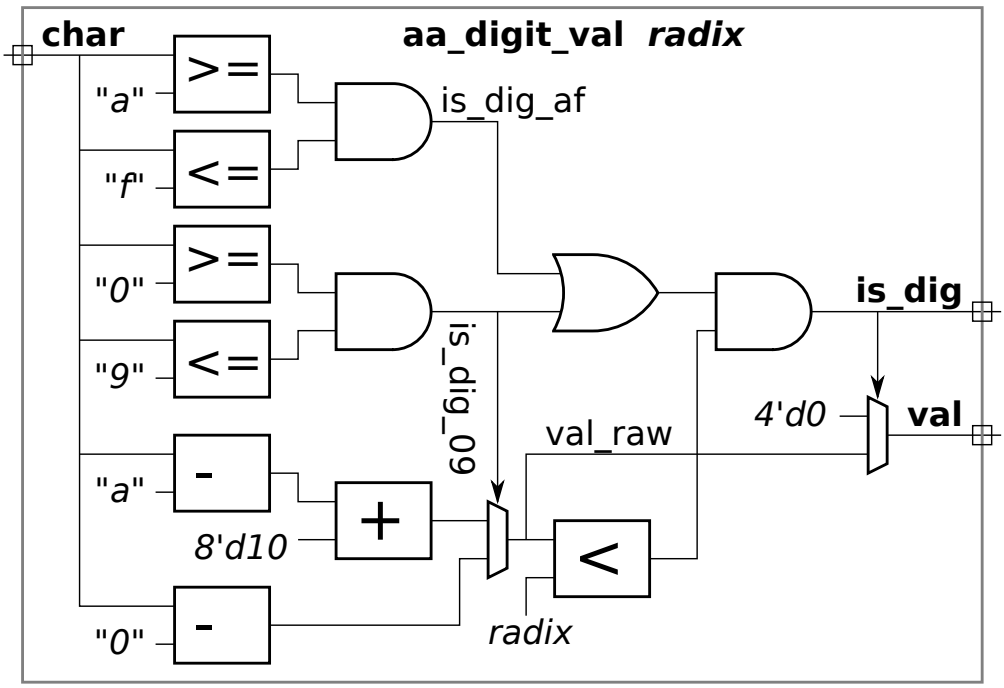
Problem 1: Module `aa_digit_val`, below, is the solution to Homework 2 Problem 1. It has an 8-bit input `char` and two outputs. Output `is_dig` is 1 iff `char` (an ASCII character) is considered a radix-R digit, where $2 \leq R \leq 16$, is the value of parameter `radix`. Output `val` is the value of that digit (in binary), or zero if it's not a digit.

```
module aa_digit_val
  #( int radix = 10 )
   ( output uwire [3:0] val,   output uwire is_dig,   input uwire [7:0] char );

   uwire  is_dig_09 = char >= "0" && char <= "9";
   uwire  is_dig_af = char >= "a" && char <= "f";
   uwire [3:0] val_raw = is_dig_09 ? char - "0" : char - "a" + 10;
   assign     is_dig = ( is_dig_09 || is_dig_af ) && val_raw < radix;
   assign     val = is_dig ? val_raw : 0;

endmodule
```

Provide sketches of what you expect the inferred hardware to look like for `aa_digit_val` as described below. *Hint: Some problems in the EE 4755 2014 Final Exam dealt with numbers in ASCII representation. The optimizations requested below must go beyond those found in the exam solution.*

(*a*) Show a sketch of the inferred hardware before any optimization is done.

Solution appears below. Items in *italic* are constants.

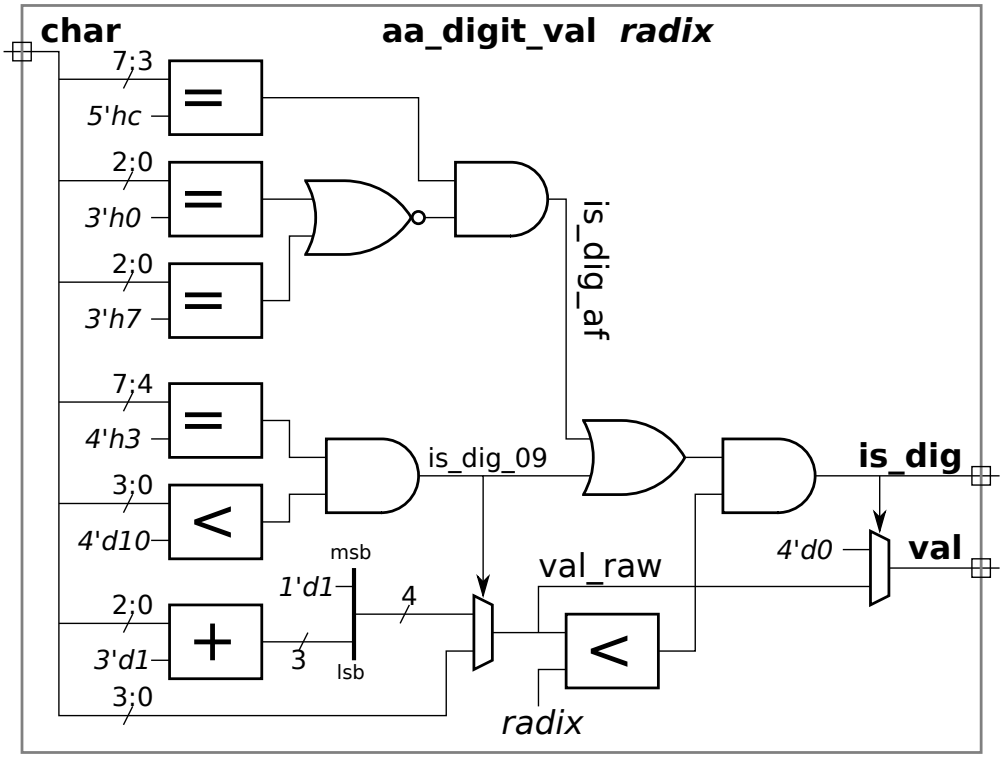


(*b*) Show a sketch of the inferred hardware after some optimization has been performed.

- The sketches must show the product of human thought (in particular, the human who's name is on the submission), not a synthesis program.
- When considering the optimizations for the logic generating `is_dig` (including the logic for `is_dig_09` and `is_dig_af`) recall that in general the cost of logic computing `a==b` is less than the cost of logic computing `a>b`.
- When considering the optimizations for the logic generating `val` think about the subtraction operations and what they actually do when `is_dig` is true. If necessary, work out examples of the subtraction by hand in hexadecimal.

Solution appears below. The optimization to avoid some magnitude comparison when computing `is_dig_09` is based on the fact that the ASCII values of characters "0" to "9" are `0x30` to `0x39` and so one can check whether the most-significant four bits are equal to `0x3` and only do a single magnitude comparison on the lower four bits. Similarly, the optimization of `is_dig_af` is based on the fact that the ASCII values of "a" to "f" are `0x61` to `0x67`, and so one can check whether the five most significant bits are 01100_2 and whether the low three bits are neither 000_2 nor 111_2.

The logic computing the value of "0" to "9" just takes the low four bits of `char`, no arithmetic is performed. The logic computing the value of "a" to "f" adds 1 to the low three bits of `char` and puts a 1 in the MSB position to make a four-bit quantity.



There is another problem on the next page!

Problem 2: Module `aa_full_adder` from Homework 2, Problem 2 adds together two digits of a radix-`R` number represented in ASCII plus a carry in. The module description from the solution appears below.

```
module aa_full_adder
  #( int radix = 10 )
   ( output uwire [7:0] sum, output uwire carry_out, output uwire is_dig_out,
     input uwire [7:0] a, b, input uwire carry_in, input uwire is_dig_in);

   uwire [3:0] val_a, val_b;
   uwire       is_dig_a, is_dig_b;

   aa_digit_val #(radix) dva(val_a, is_dig_a, a);
   aa_digit_val #(radix) dvb(val_b, is_dig_b, b);

   assign is_dig_out = is_dig_in && ( carry_in || is_dig_a || is_dig_b );
   uwire [4:0] sum_val = carry_in + val_a + val_b;
   assign     carry_out = sum_val >= radix;
   uwire [3:0] sum_dig_val = carry_out ? sum_val - radix : sum_val;
   assign sum = !is_dig_out ? " " :
          sum_dig_val < 10  ? "0" + sum_dig_val : "a" + sum_dig_val - 10;

endmodule
```

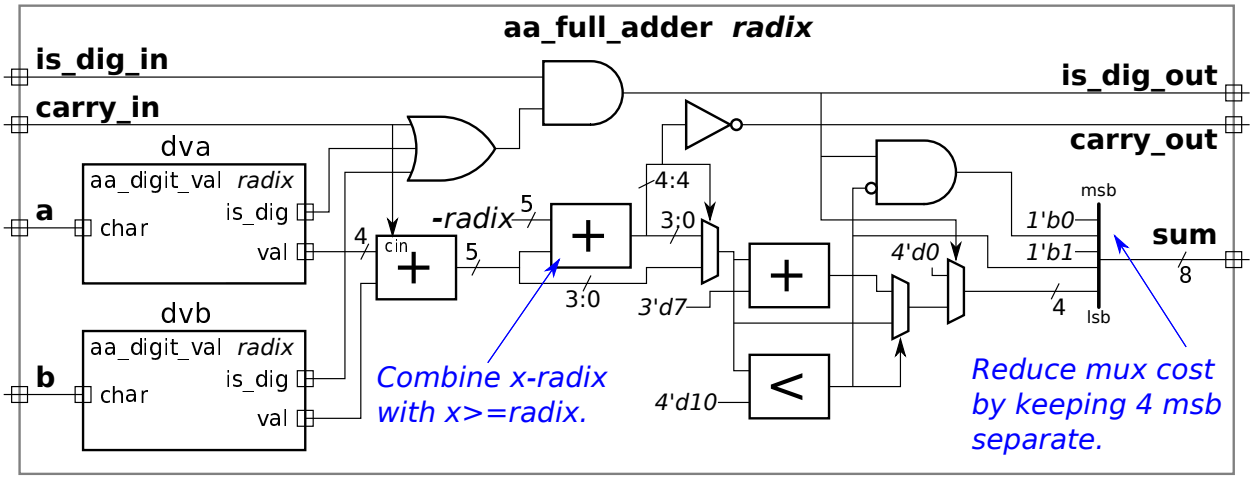
An obvious objection to an ASCII-coded radix-R adder is that it uses 8 bits to represent a digit that can be represented using only $\lceil \lg R \rceil$ bits.

(*a*) Show the hardware that might be synthesized for the module `aa_full_adder` based on the description above. This should be the inferred hardware with some optimizations applied. Take care to show the number of bits at the inputs and output of units like adders and comparison logic.

Solution appears below. Several optimizations were applied. The logic computing `sum_val >= radix` was eliminated, instead the logic computing `sum_val - radix` was widened to five bits, if the difference is positive then `sum_val >= radix` is true. If the radix is a power of two this logic would not be needed at all, an overflow can be detected by examining one bit position and `sum_val - radix` would simply be the least significant $\lg R$ bits, where R is the radix.

To save multiplexor cost, the 4 LSB of `sum` were computed separately from the 4 MSB. Note that the four possible values for the 4 MSB, `0x2` (for a space), `0x3` (for digits 0-9), and `0x6` (for digits a-f), can be easily be constructed from `is_dig_out` and `sum_dig_val<10`.

Note that only one adder is needed to compute the sum of the two digit values and the carry in, that's because the module's carry in value can go in to the adder's carry in input. It would be very wasteful to show a second adder just to add the carry in.



(*b*) Compare the cost of a d-digit ASCII-coded radix-16 adder to a $4d$-bit ripple adder. (Note that both adders can add numbers in the range of 0 to $2^{4d} - 1$.) Do so by estimating the cost in terms of the number of gates, and state any assumptions, such as the number of gates needed for an x-bit comparison unit.

The following cost model will be used. All x-input AND and OR gates have a cost of $x - 1$. Inverted inputs and outputs (those little circles) are free! Inverters are also free. A 2-input XOR cost 3 units and a 3-input XOR cost 5 units.

Based on those costs, a binary full adder cost 10 units and a n-bit ripple adder cost $10n$ units. A comparison unit can be made from a ripple adder by eliminating the sum bits, and would cost $5n$ units. An equality unit made from an XOR and an AND costs $4n$ for n bits. (The difference in cost between equality and magnitude is larger for lower-delay designs.) A w-bit, 2-input multiplexor cost $3w$ units.

In many of the adder, equality, and comparison units one of the inputs is constant. That has a big impact on cost. The cost of an n-bit ripple adder drops to $4n$ units (the BFA has a 2-input XOR and a 2-input AND gate to propagate the carry). With one input constant n-bit magnitude comparison and equality drop in cost to just n units.

When `radix` is 16, the `aa_digit_val` module will be simplified further. The `val_raw<radix` comparison is no longer necessary. Based on that the cost is `(+ 4 4 1 3 5 1 1 0 16 12 0 4); = 51` units. The `aa_full_adder` module instantiates two of these and has plenty of logic of its own. The cost including the instantiated modules is `(+ 51 51 2 1 40 4 12 12 4 1 4); = 182` units. (Figuring out the LISP syntax and attaching the costs to parts is left as an exercise to the reader.)

Based on this, the cost of a d-digit ASCII adder is $182d$ units. The equivalent ripple adder costs just $40d$ units. Sure, we expected the ASCII adder to cost more, but over $4\times$ more? Notice that a big part of the ASCII adder's cost are the two `aa_digit_val` modules, $112d$ units.

LSU EE 4755 Homework 4 Solution Due: 12 October 2016

Problem 0: First, follow the instructions for account setup and homework workflow on the course procedures page, `https://www.ece.lsu.edu/koppel/v/proc.html`.

Look through the code in `hw04.v`. Module `lookup_behav` in file `hw04.v` has a w-bit input `char` and an n-element array of w-bit quantities named `chars`. (Parameter `nelts` is n and parameter `charsz` is w.) The module also has a 1-bit output `found` which is logic `1` iff any element of `chars` is equal to `char`. Finally, the module has a $\lceil \lg n \rceil$-bit output `index` which is set to the element number of `chars` that matches `char`, or `0` if `found` is `0`. Assume that no two elements of `chars` are identical.

For example, suppose input `char` is set to `102` and that `chars` is `{63,124,102,92}`. Then output `found` will be `1` and `index` will be `2`. If `char` were `7` `index` would be `0` and `found` would be `0`, if `char` were `63` `index` would be `0` and `found` would be `1`, etc. The alert student will have recognized that $n = 4$ and that $w \geq 7$ in these examples.

Module lookup is coded in synthesizable behavioral form that describes combinational logic. The `hw04.v` file contains two other modules which are to do the same thing, `lookup_linear` and `lookup_tree`, but those modules are not yet finished.

The testbench tests all of these modules. It tests them for sizes (n) of 4, 5, 10, 15, 16, 30, 40, and 64. To change which sizes are tested (or the order in which they are tested) edit the `testbench` module.

To have the testbench test only some of these modules (say, skip the `lookup_tree` tests until after `lookup_linear` is working) look for the `for` loop with `mut=0` and modify it appropriately. (It should be easy to figure out the numbers.)

A synthesis script is provided that will synthesize all three modules at different sizes and both with and very lax timing constraint and a very strict timing constraint. The script can be run using the command `rc -files syn.tcl`. Initially it will stop with an error. To see it run to completion before starting the assignment have it only synthesize `lookup_behav` (see below). Pre-set synthesis options (in file `.synth_init`) were chosen to reject any design that is not combinational.

If there is an error when using the synthesis script then follow the manual synthesis steps on the procedures page and look for error messages.

To change which modules are synthesized edit the `set modules` line (near the bottom) in file `syn.tcl`. The values for `nelts` and other items can also be changed by editing the file.

Note: There are no points for this problem.

Problem 1: Complete `lookup_linear` so that it does the same thing as `lookup_behavioral` but by using as many copies of `lookup_elt` as it needs. That is, `lookup_linear` should use generate statements to instantiate `lookup_elt` and it should include whatever other code is needed to use these instances to compute the correct outputs.

- Behavioral or structural code can be used.
- The module must be synthesizable.
- Assume that all elements of `chars` are different.

(The complete solution Verilog code is in the assignment directory and at `https://www.ece.lsu.edu/koppel/v/2016/hw04-sol.v.html`.) There are two approaches to solving this problem. In the easy approach, which is sufficient to get full credit, generate statements are used to instantiate the `lookup_elt` modules but behavioral code is used to compute `index`.

In the alternative solution (`lookup_linear_alt`), generate statements are used both to instantiate the modules and compute index. To compute index an array of wires, `[idx_sz-1:0]idx_i[nelts-1:-1]` is declared. Element `idx_i[i]` is the value of `index` taking into account elements 0 to `i`.

```
module lookup_linear
  #( int charsz = 8,
     int nelts = 15,  // Pronounced en-elts.
     int idx_sz = $clog2(nelts) )
   ( output logic found,
     output logic [idx_sz-1:0] index,
     input uwire [charsz-1:0] char,
     input uwire [charsz-1:0] chars[nelts-1:0] );

   /// SOLUTION – Easy
   //
   //  Instantiate nelts modules, but use use behavioral code to examine
   //  their found (match) outputs.

   // Declare wires to connect to the found outputs of the instantiated modules.
   //
   uwire    [nelts-1:0] match;

   for ( genvar i=0; i<nelts; i++ )
     lookup_elt #(charsz) le(match[i],char,chars[i]);

   always_comb begin
      found = 0;
      index = 0;
      for ( int i=0; i<nelts; i++ )
        if ( match[i] ) begin index = i; found = 1; end
   end

endmodule
```

```
module lookup_linear_alt
  #( int charsz = 8,
     int nelts = 15,  // Pronounced en-elts.
     int idx_sz = $clog2(nelts) )
   ( output logic found,
     output logic [idx_sz-1:0] index,
     input uwire [charsz-1:0] char,
     input uwire [charsz-1:0] chars[nelts-1:0] );

   /// SOLUTION – Alternative
   //
   //  Use generate statements to instantiate the modules and to
   //  generate logic to find the index.


   // Instantiate nelts lookup_elt modules and compute found.
   //
   uwire    [nelts-1:0] match;

   for ( genvar i=0; i<nelts; i++ )
     lookup_elt #(charsz) le(match[i],char,chars[i]);

   assign found = | match;


   // Instantiate logic to find the index of the last matching character.
   //

   uwire [idx_sz-1:0]   idx_i[nelts-1:-1];

   assign idx_i[-1] = 0;

   for ( genvar i=0; i<nelts; i++ )
     // If no match pass along previous idx_i, otherwise replace it with i.
     assign idx_i[i] = match[i] ? i : idx_i[i-1];

   assign   index = idx_i[nelts-1];

endmodule
```

Problem 2: Complete module `lookup_tree` so that it performs the lookup using recursive instantiations of itself. Take care so that `index` is computed efficiently. *Hint: think about how to compute index efficiently when n (*`nelts`*) is a power of 2, then get the same efficiency for any n.*

If completed correctly, the cost and especially the performance at larger sizes should be better than `lookup_behavioral` and (unless you did an unexpectedly good job) better than `lookup_linear`.

- Behavioral or structural code can be used.
- The module must be synthesizable.
- Assume that all elements of `chars` are different.

(The complete solution Verilog code is in the assignment directory and at `https://www.ece.lsu.edu/koppel/v/2016/hw04-sol.v.html`.) First, we need to use generate statements to split elaboration into two cases: $n = 1$, and $n > 1$. For $n = 1$ `index` will always be zero (there's only one element in the array and its index is zero), and `found` can directly be assigned the expression `char == chars[0]` .

Two solutions will be described. In `lookup_tree_simple` work is split evenly between the two instantiated modules but this results in a more costly computation of `index` than is necessary. In `lookup_tree`, the size (value of `nelts`) of one instantiated module is forced to be a power of 2, reducing cost.

For $n > 1$ we need to split the input array, `chars`, between two instantiated `lookup_tree` modules and combine their `found` and `index` outputs. In `lookup_tree_simple` the array is split in half, the approach used in the `pop_n` module presented in class. Objects `lo_sz` and `hi_sz` are the sizes of the instantiated modules, note that these are used to compute the number of bits in the index outputs.

Logic is also needed to take the found and index outputs of the two instantiated modules, named `lo_f`, `hi_f`, `lo_idx`, and `hi_idx`, and compute the `found` and `index` outputs of the module. A mistake that many students make when trying to solve this problem is to try to take into account what is happening in all instantiated modules at every level when designing this logic. Instead, just assume that `lo_f`, `hi_f`, `lo_idx`, and `hi_idx` are correct, and use them to compute `found` and `index`. If such logic can be found, then the module will work at any size.

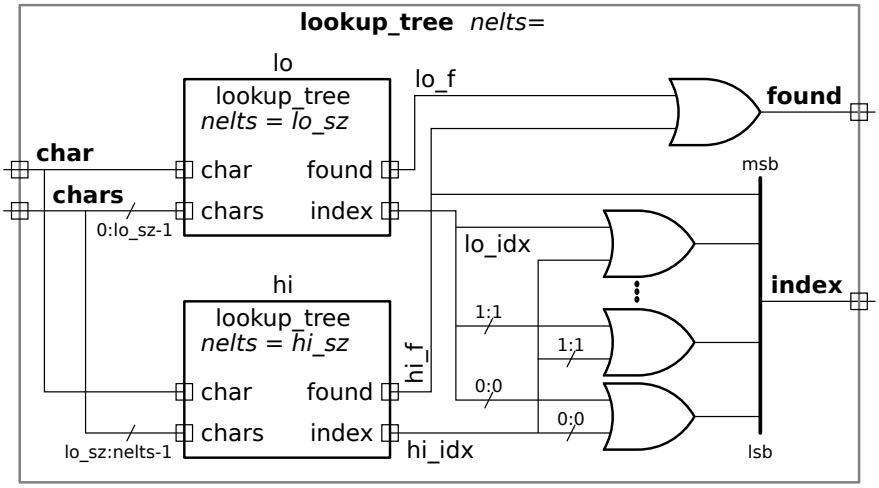
The output `found` is simply the OR of `lo_f` and `hi_f`. If `lo_f` is `1`, then `index` is `lo_idx`, but if `hi_f` is `1` then `index` is `lo_sz + hi_idx`. We don't need to worry about both `lo_f` and `hi_f` being one (the problem statement said it couldn't happen). If `hi_f` and `lo_f` are both `0` then `lo_idx` and `hi_idx` will both be `0` and `index` should be set to zero. Therefore, `index` can be set to `hi_f ? lo_sz + hi_idx : lo_idx`. That's it for the simple solution.

The problem though was to find a solution that computed `index` efficiently. Consider the sum `lo_sz + hi_idx`. If `lo_sz` were chosen to be a power of 2, and `lo_sz >= hi_sz` then instead of adding we would just be putting a `1` in bit position `lo_bits`: `{1'b1,hi_idx}`. We can re-write this as `{hi_f,hi_idx}` since this is the case where `hi_f` is `1`. And since `hi_f` is `1` we know `lo_idx` is all zeros, so we can use the expression `{ hi_f, lo_idx | hi_idx }`. As the alert student may have realized, that expression also is correct for the case where `lo_f` is `1` and the case where both are `0`. The OR gates are much less expensive than an adder and a multiplexor, even an adder with a constant input.

The code for the two modules appears below, along with the inferred hardware for the second module (that computes index efficiently.)

```
module lookup_tree_simple
  #( int charsz = 8,
     int nelts = 15,
     int idx_sz = $clog2(nelts) )
   ( output uwire found,
     output uwire [idx_sz-1:0] index,
     input uwire [charsz-1:0] char,
     input uwire [charsz-1:0] chars[nelts] );

   /// SOLUTION – Unoptimized

   if ( nelts == 1 ) begin

      assign found = char == chars[0];
      assign index = 0;

   end else begin

      // Split the character array between recursive instantiations.
      //
```

```
        localparam int lo_sz = nelts / 2;
        localparam int lo_bits = $clog2(lo_sz);
        localparam int hi_sz = nelts - lo_sz;
        localparam int hi_bits = $clog2(hi_sz);
        //
        // Note that we need to compute lo_bits and hi_bits correctly so
        // that we can declare index connections, lo_idx and hi_idx, of
        // the correct size.

        uwire       lo_f, hi_f;
        uwire [lo_bits-1:0] lo_idx;
        uwire [hi_bits-1:0] hi_idx;

        lookup_tree #(charsz,lo_sz) lo( lo_f, lo_idx, char, chars[ 0:lo_sz-1 ] );
        lookup_tree #(charsz,hi_sz) hi( hi_f, hi_idx, char, chars[lo_sz:nelts-1]);

        assign     found = lo_f || hi_f;

        assign     index = hi_f ? lo_sz + hi_idx : lo_idx;

    end

endmodule
```




```
module lookup_tree
  #( int charsz = 8,  int nelts = 15,  int idx_sz = $clog2(nelts) )
   ( output uwire found,  output uwire [idx_sz-1:0] index,
     input uwire [charsz-1:0] char,  input uwire [charsz-1:0] chars[nelts] );

   /// SOLUTION – Preferred

   if ( nelts == 1 ) begin

      assign found = char == chars[0];
      assign index = 0;  // Actually, we are assigning a zero-bit vector.

   end else begin

      // Make the size of the first lookup_tree (lo) a power of two.
      localparam int lo_bits = idx_sz - 1;
      localparam int lo_sz = 1 << lo_bits;

      // Compute the size of the second lookup_tree (hi).
      localparam int hi_sz = nelts - lo_sz;
      localparam int hi_bits = $clog2(hi_sz);

      uwire       lo_f, hi_f;
      uwire [lo_bits-1:0] lo_idx;
      uwire [hi_bits-1:0] hi_idx;

      lookup_tree #(charsz,lo_sz) lo( lo_f, lo_idx, char, chars[ 0:lo_sz-1 ] );
      lookup_tree #(charsz,hi_sz) hi( hi_f, hi_idx, char, chars[lo_sz:nelts-1]);

      assign     found = lo_f || hi_f;

      if ( lo_bits == 0 ) assign index = hi_f;
      else                assign index = { hi_f, hi_idx | lo_idx };
   end

endmodule
```

Problem 3: Run the synthesis script and characterize the strengths and weaknesses of each module. (For example, module X has lowest cost for low-speed designs.)

In a follow-on homework assignment additional questions will be asked about these modules.

The cost of the tree solution is almost always lower than the other designs, the performance is usually but not always better. For the low-cost (large delay) configurations behavioral design is usually most expensive, but is less expensive than the linear designs for the high-performance designs.

```
Note: linear_tree below is linear_tree_simple above,
  and linear_tree_opt below is linear_tree above.
```

Module Name	Area	Delay Actual	Delay Target
lookup_behav_charsz8_nelts4	9152	927	10000
lookup_linear_charsz8_nelts4	9012	990	10000
lookup_tree_charsz8_nelts4	8916	1026	10000
lookup_tree_opt_charsz8_nelts4	8988	952	10000
lookup_behav_charsz8_nelts15	35444	2348	10000
lookup_linear_charsz8_nelts15	34996	2338	10000
lookup_tree_charsz8_nelts15	34280	2606	10000
lookup_tree_opt_charsz8_nelts15	33532	2238	10000
lookup_behav_charsz8_nelts32	74648	3691	10000
lookup_linear_charsz8_nelts32	74212	3257	10000
lookup_tree_charsz8_nelts32	70932	2480	10000
lookup_tree_opt_charsz8_nelts32	71084	2443	10000
lookup_behav_charsz8_nelts40	94028	3862	10000
lookup_linear_charsz8_nelts40	94288	2585	10000
lookup_tree_charsz8_nelts40	95996	3501	10000
lookup_tree_opt_charsz8_nelts40	89292	2778	10000
lookup_behav_charsz8_nelts60	143268	5913	10000
lookup_linear_charsz8_nelts60	141792	5638	10000
lookup_tree_charsz8_nelts60	142828	3963	10000
lookup_tree_opt_charsz8_nelts60	138288	3501	10000
lookup_behav_charsz8_nelts4	12304	621	100
lookup_linear_charsz8_nelts4	13344	594	100
lookup_tree_charsz8_nelts4	13280	598	100
lookup_tree_opt_charsz8_nelts4	10888	640	100
lookup_behav_charsz8_nelts15	46896	1136	100
lookup_linear_charsz8_nelts15	47528	1120	100
lookup_tree_charsz8_nelts15	45268	1151	100
lookup_tree_opt_charsz8_nelts15	41696	1003	100
lookup_behav_charsz8_nelts32	105032	1247	100
lookup_linear_charsz8_nelts32	108688	1288	100
lookup_tree_charsz8_nelts32	96980	1093	100
lookup_tree_opt_charsz8_nelts32	96408	1056	100
lookup_behav_charsz8_nelts40	120132	1523	100
lookup_linear_charsz8_nelts40	131344	1114	100
lookup_tree_charsz8_nelts40	134444	1260	100
lookup_tree_opt_charsz8_nelts40	116320	1144	100
lookup_behav_charsz8_nelts60	184892	1726	100
lookup_linear_charsz8_nelts60	210512	1461	100

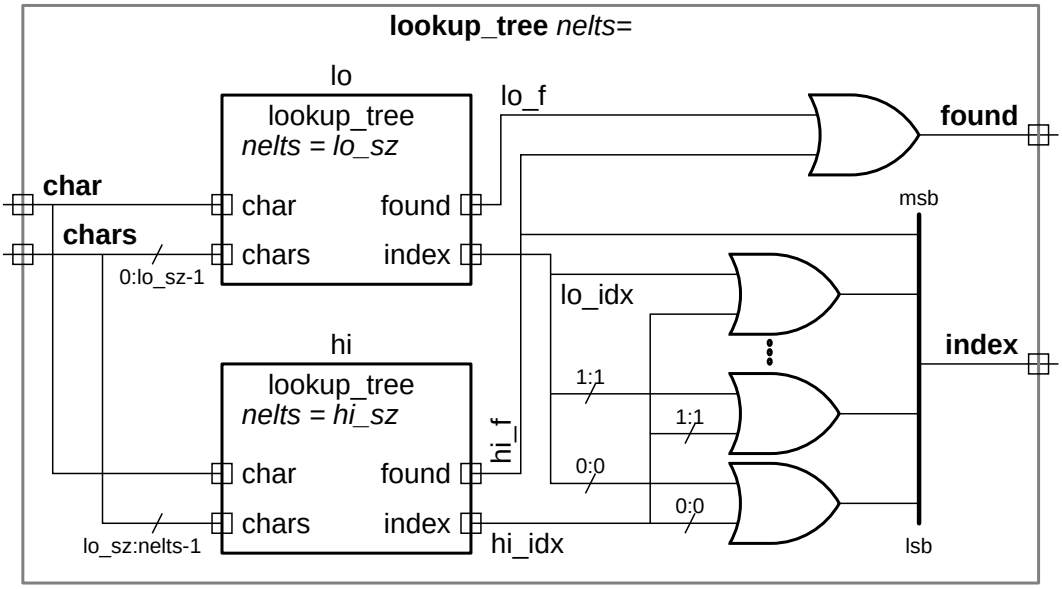
```
lookup_tree_charsz8_nelts60          185628    1890     100
lookup_tree_opt_charsz8_nelts60      176544    1500     100
```

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2016 Homework 4 --- SOLUTION
//

 /// Assignment http://www.ece.lsu.edu/koppel/v/2016/hw04.pdf
 /// Solution: http://www.ece.lsu.edu/koppel/v/2016/hw04_sol.pdf

`default_nettype none

////////////////////////////////////////////////////////////////////////////////
///  Problem 0
//
 //  Look over but don't modify this module.


module lookup_behav
  #( int charsz = 8,
     int nelts = 15,
     int idx_sz = $clog2(nelts) )
   ( output logic found,
     output logic [idx_sz-1:0] index,
     input uwire [charsz-1:0] char,
     input uwire [charsz-1:0] chars[nelts] );

   always_comb begin
      found = 0;
      index = 0;
      for ( int i=0; i<nelts; i++ )
        if ( chars[i] == char ) begin
           index = i;
           found = 1;
        end
   end

endmodule


////////////////////////////////////////////////////////////////////////////////
///  Problem 1
//
 /// Complete lookup_linear so that it does the lookup using instantiated lookup_elt.
//
//     [✔] The code must be synthesizable.
//     [✔] The code must synthesize to combinational logic. (No latches.)
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Can use behavioral or implicit structural code.
//     [✔] Do not rename modules or change ports.


module lookup_elt
  #( int charsz = 4 )
   ( output logic match,
     input uwire [charsz-1:0] char_lookup,
     input uwire [charsz-1:0] char_elt);
   /// Don't modify this module.
   always_comb match = char_lookup == char_elt;

endmodule

module lookup_linear
  #( int charsz = 8,
     int nelts = 15,  // Pronounced en-elts.
     int idx_sz = $clog2(nelts) )
   ( output logic found,
     output logic [idx_sz-1:0] index,
     input uwire [charsz-1:0] char,
     input uwire [charsz-1:0] chars[nelts-1:0] );

   /// SOLUTION -- Easy
   //
   //  Instantiate nelts modules, but use use behavioral code to examine
   //  their found (match) outputs.

   // Declare wires to connect to the found outputs of the instantiated modules.
   //
   uwire    [nelts-1:0] match;

   for ( genvar i=0; i<nelts; i++ )
     lookup_elt #(charsz) le(match[i],char,chars[i]);
```

```
   always_comb begin
      found = 0;
      index = 0;
      for ( int i=0; i<nelts; i++ )
        if ( match[i] ) begin index = i; found = 1; end
   end

endmodule

module lookup_linear_alt
  #( int charsz = 8,
     int nelts = 15,
     int idx_sz = $clog2(nelts) )
   ( output logic found,
     output logic [idx_sz-1:0] index,
     input uwire [charsz-1:0] char,
     input uwire [charsz-1:0] chars[nelts-1:0] );

   /// SOLUTION -- Alternative
   //
   //  Use generate statements to instantiate the modules and to
   //  generate logic to find the index.


   // Instantiate nelts lookup_elt modules and compute found.
   //
   uwire    [nelts-1:0] match;

   for ( genvar i=0; i<nelts; i++ )
     lookup_elt #(charsz) le(match[i],char,chars[i]);

   assign found = | match;


   // Instantiate logic to find the index of the last matching character.
   //

   uwire [idx_sz-1:0]   idx_i[nelts-1:-1];

   assign idx_i[-1] = 0;

   for ( genvar i=0; i<nelts; i++ )
     // If no match pass along previous idx_i, otherwise replace it with i.
     assign idx_i[i] = match[i] ? i : idx_i[i-1];

   assign   index = idx_i[nelts-1];

endmodule


////////////////////////////////////////////////////////////////////////////////
///  Problem 2
//
 /// Complete lookup_tree so that it does the lookup using recursive
 /// instantiations of itself.
//
//     [✔] The code must be synthesizable.
//     [✔] The code must synthesize to combinational logic. (No latches.)
//     [✔] Make sure that the testbench does not report errors.
//     [✔] Can use behavioral or implicit structural code.
//     [✔] Do not rename modules or change ports.


module lookup_tree_simple
  #( int charsz = 8,
     int nelts = 15,
     int idx_sz = $clog2(nelts) )
   ( output uwire found,
     output uwire [idx_sz-1:0] index,
     input uwire [charsz-1:0] char,
     input uwire [charsz-1:0] chars[nelts] );

   /// SOLUTION -- Unoptimized

   if ( nelts == 1 ) begin

      assign found = char == chars[0];
      assign index = 0;

   end else begin
```

```
      // Split the character array between recursive instantiations.
      //
      localparam int lo_sz = nelts / 2;
      localparam int lo_bits = $clog2(lo_sz);
      localparam int hi_sz = nelts - lo_sz;
      localparam int hi_bits = $clog2(hi_sz);
      //
      // Note that we need to compute lo_bits and hi_bits correctly so
      // that we can declare index connections, lo_idx and hi_idx, of
      // the correct size.

      uwire       lo_f, hi_f;
      uwire [lo_bits-1:0] lo_idx;
      uwire [hi_bits-1:0] hi_idx;

      lookup_tree #(charsz,lo_sz) lo( lo_f, lo_idx, char, chars[ 0:lo_sz-1 ] );
      lookup_tree #(charsz,hi_sz) hi( hi_f, hi_idx, char, chars[lo_sz:nelts-1]);

      assign     found = lo_f || hi_f;

      assign     index = hi_f ? lo_sz + hi_idx : lo_idx;
      //
      /// Notes:
      //
      //  It's okay to use lo_idx if hi_f is false, because if
      //  lo_f is false too then lo_idx must be zero.
      //
      //  This solution is less efficient because an adder is required
      //  to compute lo_sz + hi_idx. In the preferred solution lo_sz
      //  is chosen so that it is always a power of 2, avoiding the
      //  need for an addition.

   end

endmodule


module lookup_tree
  #( int charsz = 8,
     int nelts = 15,
     int idx_sz = $clog2(nelts) )
   ( output uwire found,
     output uwire [idx_sz-1:0] index,
     input uwire [charsz-1:0] char,
     input uwire [charsz-1:0] chars[nelts] );

   /// SOLUTION -- Preferred

   if ( nelts == 1 ) begin

      assign found = char == chars[0];
      assign index = 0;

   end else begin

      // Make the size of the first lookup_tree, lo, a power of two.
      //
      localparam int lo_bits = idx_sz - 1;
      localparam int lo_sz = 1 << lo_bits;

      // Compute the size of the second lookup_tree, hi.
      //
      localparam int hi_sz = nelts - lo_sz;
      localparam int hi_bits = $clog2(hi_sz);

      uwire       lo_f, hi_f;
      uwire [lo_bits-1:0] lo_idx;
      uwire [hi_bits-1:0] hi_idx;

      lookup_tree #(charsz,lo_sz) lo( lo_f, lo_idx, char, chars[ 0:lo_sz-1 ] );
      lookup_tree #(charsz,hi_sz) hi( hi_f, hi_idx, char, chars[lo_sz:nelts-1]);

      assign     found = lo_f || hi_f;

      if ( lo_bits == 0 ) assign index = hi_f;
      else                assign index = { hi_f, hi_idx | lo_idx };
      //
      /// Notes:
      //
      //  Because char can be found in at most one location and because
      //  index is zero if the char is not found, we can compute
```

```
//  index as:
//
//     index = ( hi_f ? lo_sz : 0 ) + lo_idx + hi_idx;
//
//  or even better:
//
//     index = ( hi_f ? lo_sz : 0 ) + ( lo_idx | hi_idx );
//
//  Because lo_sz is a power of two, and because lo_sz > hi_idx,
//  lo_sz > lo_idx, and lo_sz >= hi_sz we can use concatenation
//  to avoid the add:
//
//     index =  ( hi_f << lo_bits ) + ( lo_idx | hi_idx );
//     index =  { hi_f, lo_idx | hi_idx };
//
/// Example:
//
//  nelts = 14, lo_sz = 8, lo_bits = 3, hi_sz = 6, hi_bits = 3;
//  chars = { 100, 101, ... , 113 }
//
//  Suppose: char = 106.  We want index = 6.
//   lo_f = 1, lo_idx = 6 = 3'b110
//   hi_f = 0, hi_idx = 0.
//   idx = { 1'b0, 3'b110 + 3'b0 } = { 1'b0, 3'b110 } = 4'b0110 = 6
//
//  Suppose: char = 112.  We want index = 12.
//   lo_f = 0, lo_idx = 0.
//   hi_f = 1, hi_idx = 4 = 3'b100
//   idx = { 1'b1, 3'b000 + 3'b100 } = { 1'b1, 3'b100 } = 4'b1100 = 12
//
/// Synthesized Hardware:
//
```




```
//

   end

endmodule

`ifdef xxx
Module Name                              Area   Delay   Delay
                                              Actual  Target

lookup_behav_charsz8_nelts4              9152     927   10000
lookup_linear_charsz8_nelts4             9012     990   10000
lookup_tree_charsz8_nelts4               8916    1026   10000
lookup_tree_opt_charsz8_nelts4           8988     952   10000
lookup_behav_charsz8_nelts15            35444    2348   10000
lookup_linear_charsz8_nelts15           34996    2338   10000
lookup_tree_charsz8_nelts15             34280    2606   10000
lookup_tree_opt_charsz8_nelts15         33532    2238   10000
lookup_behav_charsz8_nelts32            74648    3691   10000
lookup_linear_charsz8_nelts32           74212    3257   10000
lookup_tree_charsz8_nelts32             70932    2480   10000
lookup_tree_opt_charsz8_nelts32         71084    2443   10000
lookup_behav_charsz8_nelts40            94028    3862   10000
lookup_linear_charsz8_nelts40           94288    2585   10000
lookup_tree_charsz8_nelts40             95996    3501   10000
lookup_tree_opt_charsz8_nelts40         89292    2778   10000
lookup_behav_charsz8_nelts60           143268    5913   10000
```

```
lookup_linear_charsz8_nelts60         141792    5638   10000
lookup_tree_charsz8_nelts60           142828    3963   10000
lookup_tree_opt_charsz8_nelts60       138288    3501   10000
lookup_behav_charsz8_nelts4            12304     621     100
lookup_linear_charsz8_nelts4           13344     594     100
lookup_tree_charsz8_nelts4             13280     598     100
lookup_tree_opt_charsz8_nelts4         10888     640     100
lookup_behav_charsz8_nelts15           46896    1136     100
lookup_linear_charsz8_nelts15          47528    1120     100
lookup_tree_charsz8_nelts15            45268    1151     100
lookup_tree_opt_charsz8_nelts15        41696    1003     100
lookup_behav_charsz8_nelts32          105032    1247     100
lookup_linear_charsz8_nelts32         108688    1288     100
lookup_tree_charsz8_nelts32            96980    1093     100
lookup_tree_opt_charsz8_nelts32        96408    1056     100
lookup_behav_charsz8_nelts40          120132    1523     100
lookup_linear_charsz8_nelts40         131344    1114     100
lookup_tree_charsz8_nelts40           134444    1260     100
lookup_tree_opt_charsz8_nelts40       116320    1144     100
lookup_behav_charsz8_nelts60          184892    1726     100
lookup_linear_charsz8_nelts60         210512    1461     100
lookup_tree_charsz8_nelts60           185628    1890     100
lookup_tree_opt_charsz8_nelts60       176544    1500     100
```

```
`endif

////////////////////////////////////////////////////////////////////////////////
/// Testbench Code
//
//  The code below instantiates some of the modules above,
//  provides test inputs, and verifies the outputs.
//
//  The testbench may be modified to facilitate your solution. Of
//  course, the removal of tests which your module fails is not a
//  method of fixing a broken module. (One might modify the testbench
//  so that the first tests it performs are thoe which make it easier
//  to determine what the problem is, for example, test inputs that
//  are all 0's or all 1's.)



// cadence translate_off

function automatic int min( int a, int b );
        min = a <= b ? a : b;
endfunction

module testbench();

   localparam int nelts[] = { 4, 5, 10, 15,  16, 30, 40, 64 };
   localparam int nnelts = 8;

   logic      start_done[-1:nnelts];

   for ( genvar i = 0; i < nnelts; i++ )
     testbench_sz #(nelts[i]) aa(start_done[i],start_done[i-1]);

   initial begin

      if ( nnelts != nelts.size() ) begin
        $write("Value of nnelts, %0d, different than number of elts in nelts, %0d. (See module testbench.\n",
               nnelts, nelts.size());
        $fatal(1);
      end

      start_done[-1] = 1;

   end

endmodule

module testbench_sz
  #( int nelts = 100 )
   ( output logic done, input uwire start );

   localparam int telts = nelts * 2;
   localparam int idx_sz = $clog2(nelts);
   localparam int charsz = 8;
   localparam int charmk = ( 1 << charsz ) - 1;

   localparam int num_tests = min(nelts,500);
```

```
   localparam int stride = nelts/num_tests;
   localparam int nmuts = 3;

   logic [charsz-1:0] char, chars[telts-1:0];

   uwire               found[nmuts];
   uwire [idx_sz-1:0]  idx[nmuts];
   logic              shadow_found;
   logic [idx_sz-1:0] shadow_idx;

   lookup_behav #(charsz,nelts) l00(found[0],idx[0],char,chars[nelts-1:0]);
   lookup_linear #(charsz,nelts) l01(found[1],idx[1],char,chars[nelts-1:0]);
   lookup_tree #(charsz,nelts) l02(found[2],idx[2],char,chars[nelts-1:0]);

   string mutnames[] = { "lookup_behav", "lookup_linear", "lookup_tree" };

   int err[nmuts];

   initial begin

      automatic int tot_errors = 0;
      localparam int gap = charmk / telts;
      chars[0] = {$random} % gap;
      for ( int i=1; i<telts; i++ )
        chars[i] = chars[i-1] + 1 + {$random} % gap;
      for ( int i=0; i<telts; i++ ) begin
         automatic int idx = {$random} % telts;
         {chars[i],chars[idx]} = {chars[idx],chars[i]};
      end

      wait ( start == 1 );

      for ( int i=0; i<num_tests; i++ ) begin

         automatic int idx_try = {$random} % telts;
         char = chars[idx_try];
         shadow_found = idx_try < nelts;
         shadow_idx = idx_try;
         #1;

         for ( int mut=0; mut<3; mut++ ) begin

            automatic int cr_fnd = shadow_found === found[mut];
            automatic int cr_idx = shadow_idx === idx[mut];

            if ( cr_fnd && ( shadow_found == 0 || cr_idx ) ) continue;
            if ( err[mut] > 100 ) break;

            $write("Mod %s nelts %0d test %3d char %h: wrong %s. Found %h%s%h (correct) idx %4d %s %4d (correct)\n",
                   mutnames[mut], nelts, i, char,
                   cr_idx ? "found" : "index",
                   found[mut],
                   cr_fnd ? "==" : "!=",
                   shadow_found,
                   idx[mut], cr_fnd ? "!=" : "??", shadow_idx);

            err[mut]++;

         end

      end
      for ( int i=0; i<num_tests; i++ ) tot_errors += err[i];
      $write("For nelts %0d performed %0d tests, %0d errors found.\n",
             nelts, num_tests, tot_errors);

      done = 1;
   end

endmodule

// cadence translate_on
```

LSU EE 4755 Homework 5 Solution Due: 7 November 2016

Problem 0: This first problem provides background on the module used in this assignment. Please read the background and then solve the problems further below. The Verilog source can be found in directory `hw05`, however for this assignment there is no need to do anything with it.

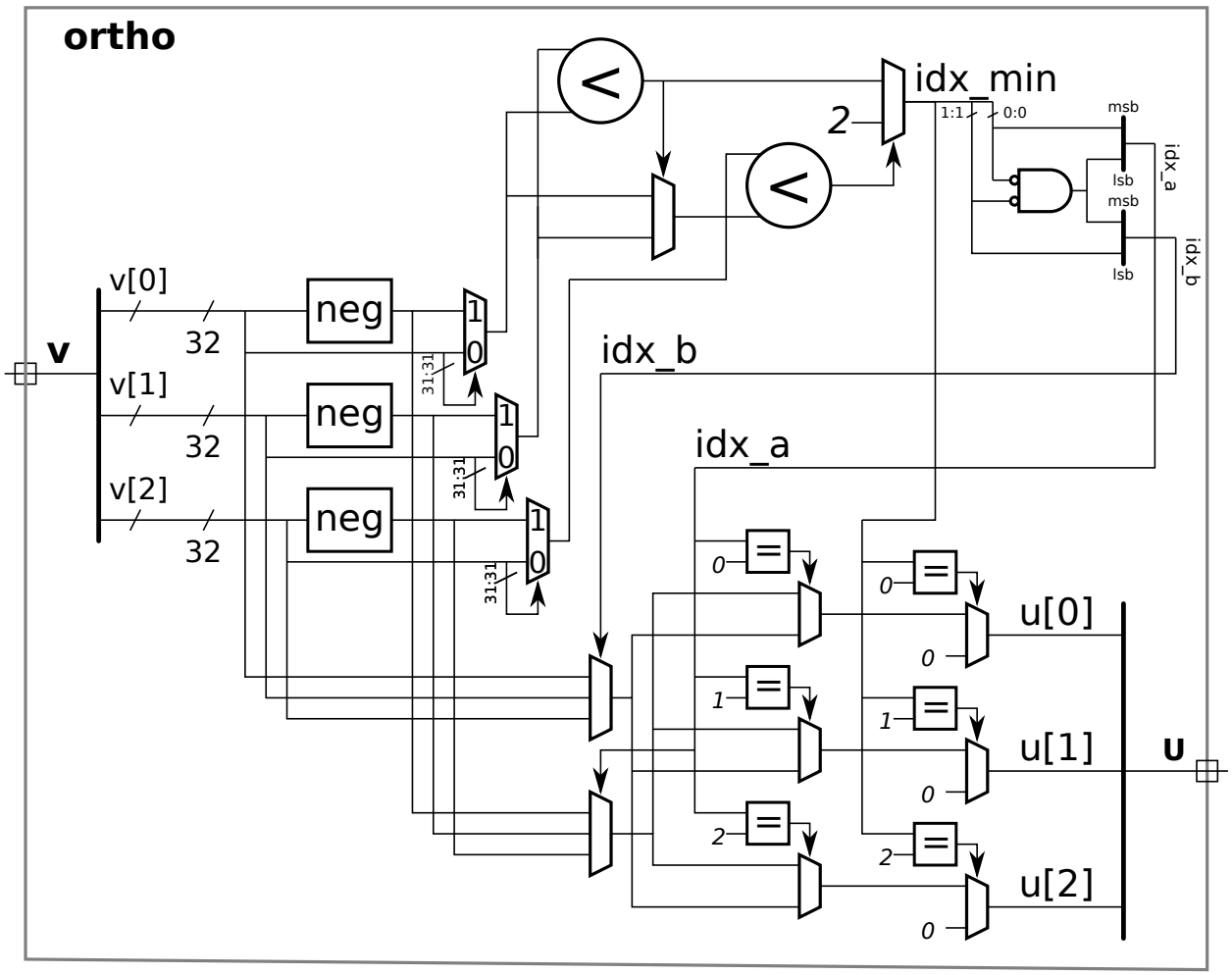
Module `ortho` has one input, `v`, a three-element vector of signed integers, and one output, `u`, also a three-element vector of signed integers. The output is computed so that `u` is orthogonal to `v` in the geometric sense. For those who are rusty on linear algebra, non-zero vectors u and v are orthogonal if $u \cdot v = 0$ or $u_x v_x + u_y v_y + u_z v_z = 0$. Using Verilog notation, `u` is computed so that `u[0]*v[0]+u[1]*v[1]+u[2]*v[2]=0` and at least one element of `u` is not zero. It does so by finding the smallest element of `v`, setting the corresponding element in `u` to zero, swapping the to remaining two elements, and negating one of the two. For example, if $v = (4, 7, 55)$ then the module would set $u = (0, 55, -7)$.

```
module ortho #( int alternative = 1, int w = 32 )
   ( output logic signed [w-1:0] u [3],   input wire signed [w-1:0] v [3] );

   logic [1:0] idx_min, idx_a, idx_b;

   always_comb begin

      idx_min = 0;
      for ( int i=1; i<3; i++ ) if ( $abs(v[i]) < $abs(v[idx_min]) ) idx_min = i;

      idx_a = ( idx_min + 1 ) % 3;
      idx_b = ( idx_min + 2 ) % 3;

      if ( alternative == 1 ) begin

         // The loop below is a hint to synthesis program Cadence Encounter 14.28.
         for ( int i=0; i<3; i++ ) u[i] = 0;

         u[idx_min] = 0;
         u[idx_a] = v[idx_b];
         u[idx_b] = -v[idx_a];

      end else if ( alternative == 2 ) begin

         for ( int i=0; i<3; i++ )
           u[i] = idx_min == i ? 0 : idx_a == i ? v[idx_b] : -v[idx_a];

      end else $fatal(1);

   end

endmodule
```

Important: For all problems below in which hardware is shown:

- Clearly show inputs and outputs of `ortho`.
- Try to draw diagrams showing all hardware for `ortho` and refer to parts of the diagram in your answers below.

Complete solution appears below. See the problems for detail.



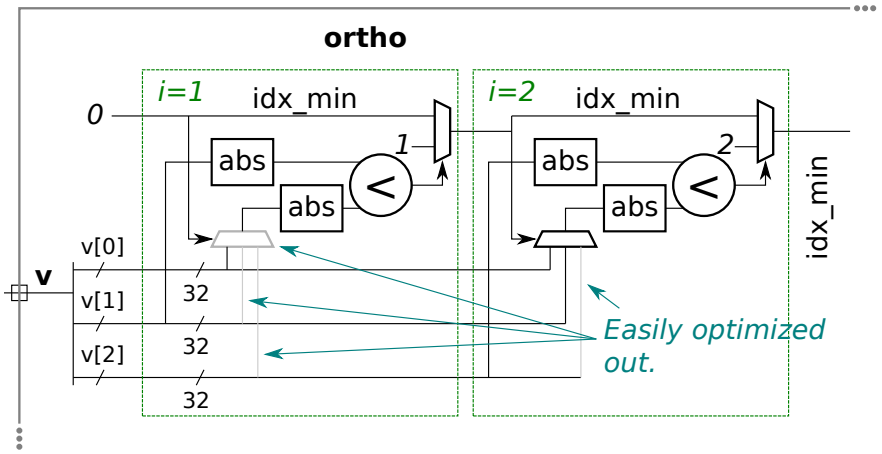
Problem 1: Consider the following part of the module:

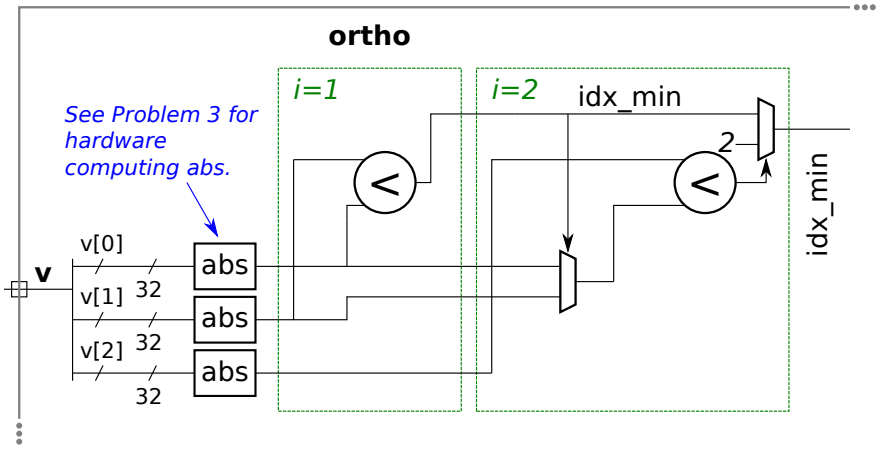
```
idx_min = 0;
for ( int i=1; i<3; i++ )
  if ( $abs(v[i]) < $abs(v[idx_min]) ) idx_min = i;
```

(*a*) Show the hardware that will be synthesized for this fragment. (Please refer to the entire module when determining what will be synthesized.) Make reasonable optimizations. (See the next subproblem.) In this subpart show `abs` as a box.

Un-optimized and optimized solution appears below. In the un-optimized solution absolute value units appear at the output of the index operation multiplexors (the multiplexors implementing `v[idx_min]`), whereas in the optimized

solution the absolute value is computed earlier. In the optimized version one index operation mux is removed entirely, in the other an input is eliminated. (See the midterm exam solution.) As shown in Problem 3, the absolute value hardware is shared with the hardware used for negation.





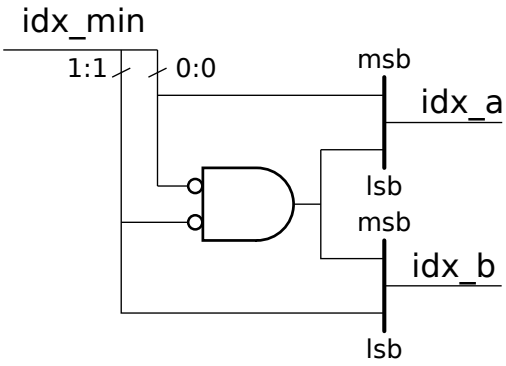
(*b*) The synthesis program synthesizes hardware that contains four absolute value units for this code, even with effort set to high. Explain why four is too many, perhaps by referring your own version that uses fewer absolute value units.

See the solution to the part above.

Problem 2: Consider the part of the module below: Show the hardware that will be synthesized for this code, taking into consideration that `idx_min` is two bits. *Hint: This is easy. Just consider all possible values of* `idx_min`.

```
idx_a = ( idx_min + 1 ) % 3;
idx_b = ( idx_min + 2 ) % 3;
```

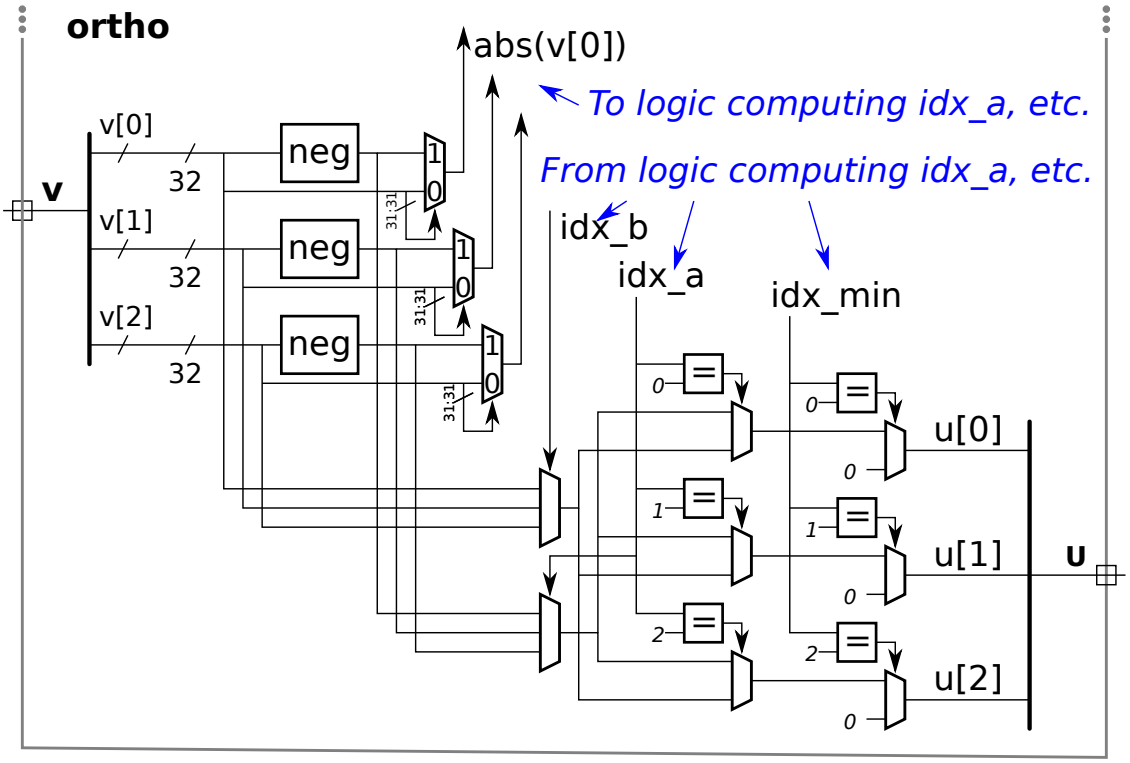
Solution appears below. The most important point is that there is no hardware to compute the remainder (modulo), which would be costly, nor are there adders. Drawing a truth table will show that only a single gate is needed.



Problem 3: Show the hardware that will be synthesized for the alternative 2 code, below, after optimization. As with the other problems, take into account the rest of the module. Look for opportunities to optimize `-v[idx_a]` taking advantage of hardware for `abs`.

```
for ( int i=0; i<3; i++ )
  u[i] = idx_min == i ? 0 : idx_a == i ? v[idx_b] : -v[idx_a];
```

Solution appears below. Since this part needs negation (computing $-x$) and the hardware computing `idx_min` needs absolute value, which uses negation, this part computes the absolute value. Negation itself of a 2's complement value is computed by negating the bits and adding one. If the negated value were only needed for an adder, or adder-like hardware, then the adder could be eliminated.



Problem 4: As directed below, estimate the critical path in `ortho` for a w-bit instantiation. Do so using ripple-adder like implementations for absolute value, comparison, and negation. Use the performance model in which n-input AND and OR gates have delay $\lceil \lg n \rceil$ units.

(a) Find the critical path using the assumption that in hardware for an expression like $a + b < c$ the delay through the adder must be added to the delay through the comparison unit. The answer should be a function of w.

Solution appears in the diagram below in the upper timing number. See the last part for details.

(b) Find the critical path accounting for the fact that in ripple-like hardware for an expression like $a + b < c$ the low bits of the comparison can start as soon as the low bits of the sum are available. The answer should be a function of w.

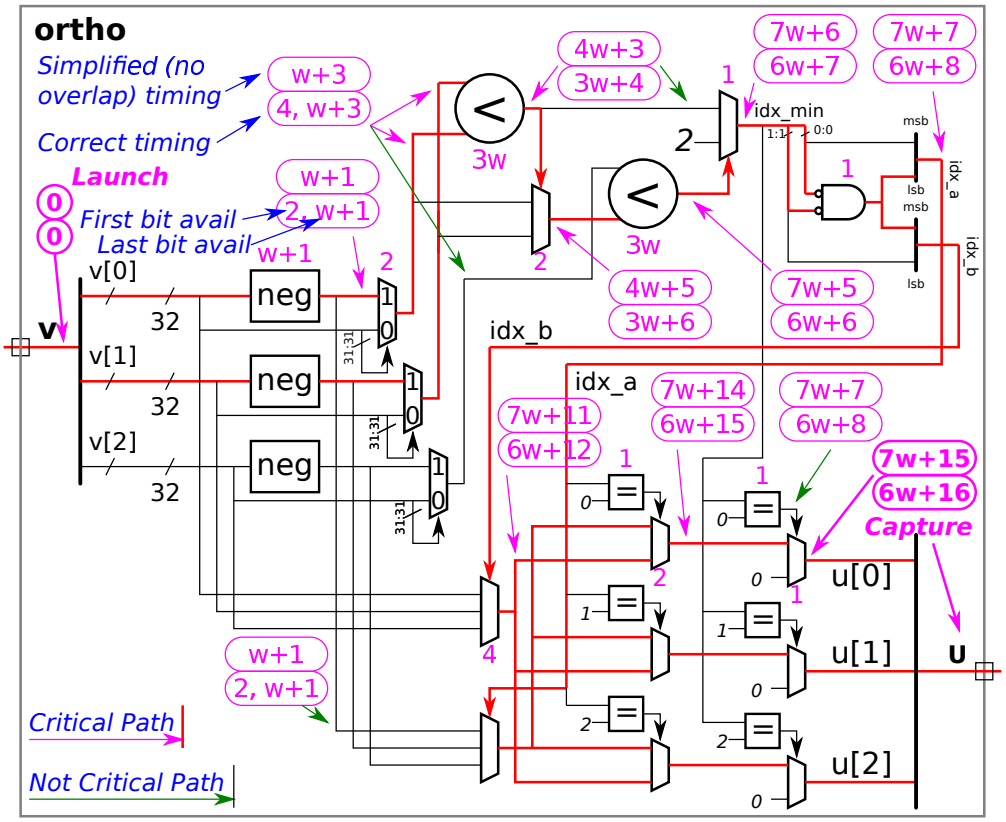
Solution appears in the diagram below in the lower timing number. See the last part for details.

(c) Show a sketch of the hardware with an arrow tracing the critical path through the hardware, from input to output. Annotating that arrow with intermediate delays will help in assigning partial credit.

The critical path appears in red in the figure below, the critical path (but not its length) is the same with both timing assumptions. The paired purple boxed numbers give the absolute time that the signal arrives at the labeled wire. The upper of the pair is under the assumption that one piece of ripple-like hardware must completely finish before a subsequent piece of ripple-like hardware can start. The lower number is computed under the correct assumption, that computation starts when data arrives. In the diagram this only affects the first comparison unit.

The delay of each component is shown as an unboxed purple number. The delay of the neg unit is based on a ripple adder constructed with binary half adders. The carry chain consists only of AND gates.

Purple arrows point to wires carrying the critical path, green arrows point to non-critical wires.

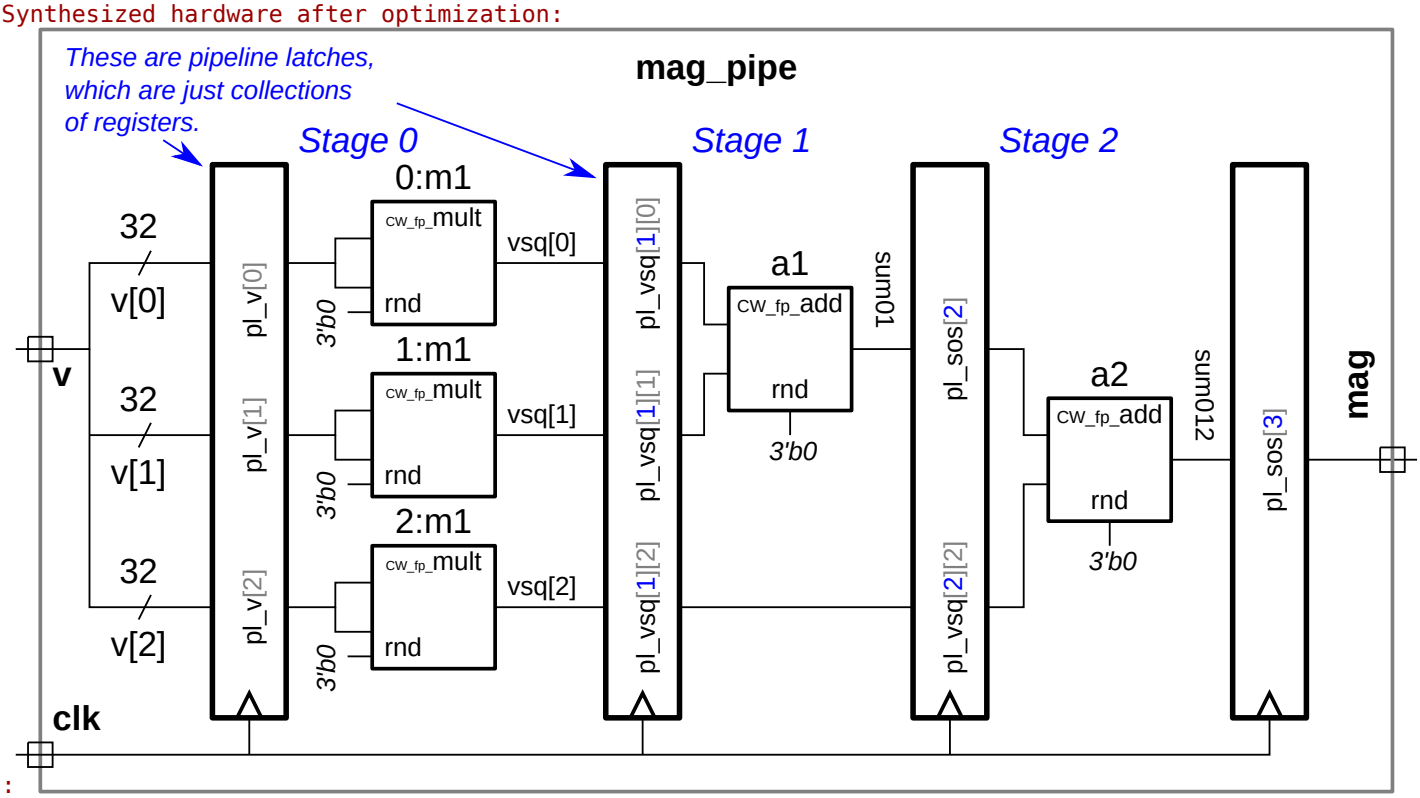


```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2016 Homework 6
/// PRELIMINARY SOLUTION
//

 /// Assignment http://www.ece.lsu.edu/koppel/v/2016/hw06.pdf

 /// Additional Resources
  //
  // Instructions for Account Setup, Verilog, Synthesis, Chipware, Emacs.
  //      http://www.ece.lsu.edu/koppel/v/proc.html
  //
  //
  // Verilog Documentation
  //    The Verilog Standard
  //      http://standards.ieee.org/getieee/1800/download/1800-2012.pdf
  //    Introductory Treatment (Warning: Does not include SystemVerilog)
  //      Brown & Vranesic, Fundamentals of Digital Logic with Verilog, 3rd Ed.
  //
  // ChipWare Component Library Documentation
  //    Documentation for the FP modules (and other) such as CW_fp_add.
  //    Look for the link to ChipWare on: http://www.ece.lsu.edu/v/ref.html
  //

// Load Verilog for ChipWare floating-point multiply and add modules.
//
`include "/apps/linux/cadence/RC142/share/synth/lib/chipware/sim/verilog/CW/CW_fp_mult.v"
`include "/apps/linux/cadence/RC142/share/synth/lib/chipware/sim/verilog/CW/CW_fp_add.v"

`default_nettype none

////////////////////////////////////////////////////////////////////////////////
///  Problem 0
//
 //  Look over but don't modify these modules.


// cadence translate_off

 /// Non-Synthesizable Mag Module --- Complete, Don't Edit
//
module mag_functional
  ( output shortreal mag,
    input shortreal v [3] );

   always_comb begin
      shortreal sos;
      sos = 0;
      for ( int i=0; i<3; i++ ) sos += v[i] * v[i];
      mag = sos;
   end

endmodule
// cadence translate_on

 /// Combinational Module  --- Complete, Don't Edit
//
module mag_comb
  ( output uwire [31:0] mag,
    input uwire [31:0] v [3] );

   uwire [31:0]   vsq[3];
   uwire [7:0]    status[5];
   uwire [31:0]    sum01;
   localparam    logic [2:0] rnd = 0; // 0 is round toward even.

   for ( genvar i=0; i<3; i++ )
     CW_fp_mult m1( v[i], v[i], rnd, status[i], vsq[i]); // Product is last!

   CW_fp_add a1( vsq[0], vsq[1], rnd, sum01, status[3] );
   CW_fp_add a2( sum01, vsq[2], rnd, mag, status[4] );
```

```
endmodule


////////////////////////////////////////////////////////////////////////////////
///  Problem 1
//
 ///  Complete mag_seq so that it computes mag sequentially, using one
 ///  fp add and one fp multiply module.
//
//      [x] Learn to use SimVision *before* wasting hours on simple problems.
//      [x] The code must be synthesizable.
//      [x] Make sure that the testbench does not report errors.
//      [x] Can use behavioral or implicit structural code.
//      [x] Do not rename modules or change ports.
//      [x] Must use exactly one CW_fp_add and one CW_fp_mult.
//      [x] Assume that data arrives at module inputs late in the clock cycle.

// cadence translate_off
class Debug;

   int cycle;
   int test_cyc;  // Number of cycles since test began.
   int test_num;
   shortreal vr[3];
   logic [31:0] v[3];
   shortreal magr; // Correct result.
   logic [31:0] mag; // Correct result.

endclass
// cadence translate_on

module mag_seq
  ( output uwire [31:0] mag,
    output uwire ready,
    input uwire [31:0] v [3],
    input uwire start,
    input uwire clk );

   // cadence translate_off
   Debug db;
   // cadence translate_on

   localparam logic [2:0] rnd = 0; // 1 is round towards zero.

   uwire [7:0] sm, sa;

   logic [31:0] accum[2];
   uwire [31:0]  prod, sum;
   logic [2:0]  step;

   ///  SOLUTION -- Assign multiplier input.
   //
   uwire [31:0]  ma = v[ step ];

   CW_fp_mult m1( .a(ma),       .b(ma),       .rnd(rnd), .z(prod), .status(sm));
   CW_fp_add  a1( .a(accum[0]), .b(accum[1]), .rnd(rnd), .z(sum),  .status(sa));

   localparam int last_step = 4;
   assign     ready = step == last_step;

   always_ff @( posedge clk )
     if ( start ) step <= 0;
     else if ( step < last_step ) step <= step + 1;

   always_ff @( posedge clk )
     begin
         case ( step )
           0: accum[0] <= prod;  // Save v[0] * v[0].

           ///  SOLUTION below.
           1: accum[1] <= prod;  // Save v[1] * v[1].

           2: begin
```

```
            accum[0] <= prod;  // Save v[2] * v[2].
            accum[1] <= sum;   // Save (v[0]*v[0]) + (v[1]*v[1])
          end

          3: accum[1] <= sum;   // Save (v[0]*v[0]+v[1]*v[1]) + (v[2]*v[2]).

        endcase
    end

  assign mag = accum[1];

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Problem 2
//
 /// Complete mag_pipe so that it computes mag in pipelined fashion and
 /// has at most one fp operation delay per cycle.
//
//     [x] Learn to use SimVision *before* wasting hours on simple problems.
//     [x] The code must be synthesizable.
//     [x] Make sure that the testbench does not report errors.
//     [x] Can use behavioral or implicit structural code.
//     [x] Do not rename modules or change ports.
//     [x] Choose number of stages to maximize throughput (minimize delay).
//     [x] Use as many CW_fp_add and CW_fp_mult modules as needed, but no more.
//     [x] Assume that data arrives at module inputs late in the clock cycle.


module mag_pipe
  ( output uwire [31:0] mag,
    input uwire [31:0] v [3],
    input uwire clk );

  // cadence translate_off
  Debug db;
  // cadence translate_on

  /// Do not rename nstages. The testbench examines its value and it must be set
  ///  correctly.
  //  For a vector arriving at cycle t, magnitude will be available at
  //  cycle t + nstages.
  localparam int nstages = 4;

  localparam logic [2:0] rnd = 0; // 1 is round towards zero.

  logic [31:0] pl_vsq[1:2][3];
  logic [31:0] pl_sos[2:3];
  uwire [31:0]  vsq[3], sum01, sum012;

  uwire [7:0]   s[5];

  // Pipeline latches between inputs and stage 0.
  //
  logic [31:0] pl_v[3];

  ///
  /// Logic Within Stages
  ///

  // Stage 0: Three Multipliers.
  //
  // Instantiate 3 multipliers. All of these are in stage 0.
  //
  for ( genvar i=0; i<3; i++ )
    CW_fp_mult m1(.a(pl_v[i]), .b(pl_v[i]),
                  .rnd(rnd), .z(vsq[i]), .status(s[i]));

  // Stage 1: An adder.
  //
  CW_fp_add a1( pl_vsq[1][0], pl_vsq[1][1], rnd, sum01, s[3] );

  // Stage 2: Another adder.
```

```
    //
    CW_fp_add a2( pl_sos[2], pl_vsq[2][2], rnd, sum012, s[4] );

    ///
    /// Pipeline Latches (Registers Separating Stages)
    ///
    always_ff @( posedge clk ) begin

       // Module input -> Stage 0
       //
       pl_v <= v;

       // Stage 0 -> 1
       //
       //  Result of multiplications done in stage 0.
       //
       pl_vsq[1] <= vsq;   // Note: vsq is a 3-element array of 32-bit vals.

       // Stage 1 -> 2
       //
       //  Pass along multiplications done in stage 1.
       //
       pl_vsq[2][2] <= pl_vsq[1][2];
       //
       //  Sum performed in stage 1.
       //
       pl_sos[2] <= sum01;

       // Stage 2 -> 3
       //
       //  Sum performed in stage 2.
       //
       pl_sos[3] <= sum012;
    end

    assign mag = pl_sos[3];

endmodule

 // Synthesized hardware after optimization:
```




```
 // :

////////////////////////////////////////////////////////////////////////////////
```

/// Testbench Code

```
//
//  The code below instantiates some of the modules above,
//  provides test inputs, and verifies the outputs.
//
//  The testbench may be modified to facilitate your solution. Of
//  course, the removal of tests which your module fails is not a
//  method of fixing a broken module. (One might modify the testbench
//  so that the first tests it performs are thoe which make it easier
//  to determine what the problem is, for example, test inputs that
//  are all 0's or all 1's.)



// cadence translate_off

function automatic real rand_real(real minv, real maxv);
      rand_real = minv + ( maxv - minv ) * ( real'({$random}) ) / 2.0**32;
endfunction

function automatic shortreal fabs(shortreal val);
      fabs = val < 0 ? -val : val;
endfunction

program reactivate
   (output uwire clk_reactive, output int cycle_reactive,
    input uwire clk, input var int cycle);
   assign clk_reactive = clk;
   assign cycle_reactive = cycle;
endprogram


module testbench();

   typedef enum { MT_comb, MT_seq, MT_pipe } Module_Type;

   localparam int wid = 32;
   localparam int max_latency = 10;
   localparam int num_tests = 16;
   localparam int nmuts = 10;
   int err[nmuts];

   uwire [31:0]  mag[nmuts];
   uwire         ready[nmuts];
   shortreal   magr;
   shortreal vr[3];
   logic [31:0] v[3];
   logic [31:0] vp[3];
   logic        start;

   typedef struct
     {
      int idx;
      int err_count = 0;
      int ncyc = 0;
      Module_Type mt = MT_comb;
      logic [wid-1:0] sout = 'h111;
      int cyc_tot = 0;
      int latency = 0;
      } Info;
   Info pi[string];

   localparam int cycle_limit = num_tests * max_latency * 4;
   int cycle, cyc_start;
   bit done;
   logic clock;
   bit   use_others;

   logic      clk_reactive;
   int cycle_reactive;
   reactivate ra(clk_reactive,cycle_reactive,clock,cycle);

   task pi_seq(input int idx, input string name);
      automatic string m = $sformatf("%s", name);
```

```
      pi[m].idx = idx; pi[m].mt = MT_seq;
   endtask

   task pi_pipe(input int idx, input string name, input int ncyc);
      automatic string m = $sformatf("%s", name);
      pi[m].idx = idx; pi[m].mt = MT_pipe;
      pi[m].ncyc = ncyc;
   endtask

   Debug db;
   initial db = new;

   initial begin
      clock = 0;
      cycle = 0;

      fork
         forever #10 begin
            cycle += clock++;
            db.cycle = cycle;
            db.test_cyc = cycle - cyc_start;
         end
         wait( done );
         wait( cycle >= cycle_limit )
           $write("*** Cycle limit exceeded, ending.\n");
      join_any;

      $finish();
   end

   mag_functional mf( magr, vr );
   mag_comb m1( mag[0], v );
   initial pi["Comb."].idx = 0;
   mag_seq m2( mag[1], ready[1], v, start, clock );
   initial begin pi_seq(1,"Seq."); m2.db = db; end
   mag_pipe m4( mag[3], vp, clock );
   initial begin pi_pipe(3,"Pipe",m4.nstages); m4.db = db; end

   initial begin

      while ( !done ) @( posedge clk_reactive ) #2

         if ( use_others ) begin

            vp = v;
            use_others = 0;
            start = 1;

         end else begin

            vp[0] = $shortrealtobits(shortreal'(cycle-cyc_start));
            vp[1] = cycle - cyc_start;
            vp[2] = 0;
            start = 0;

         end
   end

   initial begin

      automatic int tot_errors = 0;

      done = 0;
      use_others = 0;
      start = 0;

      @( posedge clk_reactive );


      for ( int i=0; i<num_tests; i++ ) begin

         automatic int awaiting = pi.num();

         db.test_num = i;
```

```
cyc_start = cycle;
db.test_cyc = 0;

if ( i < 8 ) begin

   // In first eight test vector components are zero or one.
   //
   for ( int j=0; j<3; j++ ) vr[j] = i & 1 << j ? 1.0 : 0.0;

end else begin

   // In other tests vector components are randomly chosen.
   //
   for ( int j=0; j<3; j++ ) vr[j] = rand_real(-10,+10);

end

for ( int j=0; j<3; j++ ) v[j] = $shortrealtobits(vr[j]);
db.vr = vr;
db.v = v;
fork
   #0 begin
      db.magr = magr;
      db.mag = $shortrealtobits(magr);
   end
join_none

vp = v;
use_others = 1;

/// Collect Result (mag) From Each Module Under Test (mut)
//
foreach ( pi[muti] ) begin

   automatic string mut = muti;  // Informal name of module.
   automatic Info p = pi[mut];

   // Create a child thread to get response from current mut.
   // The parent thread, without delay, proceeds to join_none.
   //
   fork begin

      automatic int steps = pi[mut].ncyc;
      automatic int latency =
       pi[mut].mt == MT_comb ? 1 :
       pi[mut].mt == MT_seq ? 2 : steps;

      // Compute time at which result should be ready or
      // when to start examining a READY output.
      //
      automatic int eta = 1 + cyc_start + latency;

      pi[mut].latency = latency;

      // Wait (just this thread waits) until result should be ready.
      //
      wait ( cycle_reactive == eta );

      // If this module has a READY output, wait for it.
      //
      if ( pi[mut].mt == MT_seq ) wait( ready[pi[mut].idx] );

      // Decrement count of the number of modules we are waiting for.
      //
      awaiting--;

      // Store the module MAG output, it will be checked later
      // for correctness.
      //
      pi[mut].sout = mag[pi[mut].idx];

      pi[mut].cyc_tot += cycle - cyc_start;

      // This thread ends execution here.
```

```
            end join_none;

         end

         // Wait until data collected from all modules under test.
         //
         wait ( awaiting == 0 );

         // Check the output of each Module Under Test.
         //
         foreach ( pi[ mut ] ) begin

            // Assign module output to a shortreal.
            //
            automatic shortreal mmagr = $bitstoshortreal(pi[mut].sout);
            //
            // Note: pi[mut].sout is type logic which is assumed to be
            // an unsigned integer. However, the contents is really an
            // IEEE 754 single-precision float (shortreal in
            // SystemVerilog) and so $bitstoshortreal is used so that
            // pi[mut].sout is copied bit-for-bit unchanged to mmagr.

            // Compute difference between module output and expected
            // output.  With FP small differences can be okay, they might
            // occur, for example, due to differences in the order of
            // operations.
            //
            automatic shortreal err_mag = fabs( mmagr - magr );
            automatic bit okay = err_mag < 1e-4;

            if ( !okay ) begin
              pi[mut].err_count++;
              if ( pi[mut].err_count < 5 )
                $write("%s test #%0d vec (%.1f,%.1f,%.1f) error: h'%8h  %7.4f != %7.4f (correct)\n",
                   mut, i, vr[2], vr[1], vr[0],
                       pi[mut].sout, mmagr, magr);
            end
         end

         while ( {$random} & 1 == 1 ) @( posedge clk_reactive );
         //
         // Note: By waiting for reactive clock we can be sure that
         // modules under test have completed all work due to the
         // positive edge of the regular clk. Wait a random amount of
         // time in case any modules are only correct at some stride.

      end

      foreach ( pi[ mut ] )
        $write("Ran %4d tests for %-25s, %4d errors found. Avg cyc %.1f\n",
               num_tests, mut, pi[mut].err_count,
               pi[mut].mt == MT_comb ? 1 : real'(pi[mut].cyc_tot) / num_tests);

      done = 1;

      $finish(2);

   end

endmodule

// cadence translate_on
```

23 Fall 2015 Solutions

LSU EE 4755 Homework 1 Solution Due: 9 September 2015

The questions below can be answered without using EDA software, paper and pencil will suffice. Please turn in the solution on paper. Homework 2 will require the use of Verilog implementations.

Those who are rusty about the correspondence between Verilog code and hardware might want to look at the solution to EE 3755 Fall 2013 Homework 1, at `http://www.ece.lsu.edu/ee3755/2013f/hw01_sol.pdf`.

Problem 1: The routine `shift_right_fixed_amt` uses the `>>` operator to perform the right shift. Perhaps you are wondering if the operation is an arithmetic right shift or a logical right shift. (In a logical right shift the vacated bit positions are always set to zero, in an arithmetic shift they are set to the MSB of the input.) Look up the operation performed by this operator in the SystemVerilog 2012 documentation.

```
module shift_right_fixed_amt
  #( int fsamt = 4 )   // Fixed shift amount.
   ( output wire [15:0] shifted,
     input wire [15:0] unshifted,
     input wire shift );

   // If shift is true shift by fsamt, otherwise don't shift.
   //
   assign    shifted =  shift  ?  unshifted >> fsamt  :  unshifted;

endmodule
```

(*a*) Indicate the section and page in which this information can be found.

Section 11.4.10, on page 233.

(*b*) Show how the module can be modified to perform the other kind of shift (if it's currently arithmetic, make it logical, if it's currently logical make it arithmetic).

Two changes need to be made: The type of the value to be shifted must be changed to signed, and the operator must be changed from `>>` to `>>>`. The changed code appears below.

```
module shift_right_fixed_amt_sol
  #( int fsamt = 4 )   // Fixed shift amount.
   ( output wire [15:0] shifted,
     input wire signed [15:0] unshifted,  // SOLUTION, change to signed.
     input wire shift );

   // SOLUTION, change ">>" operator to ">>>".
   //
   assign    shifted =  shift  ?  unshifted >>> fsamt  :  unshifted;

endmodule
```

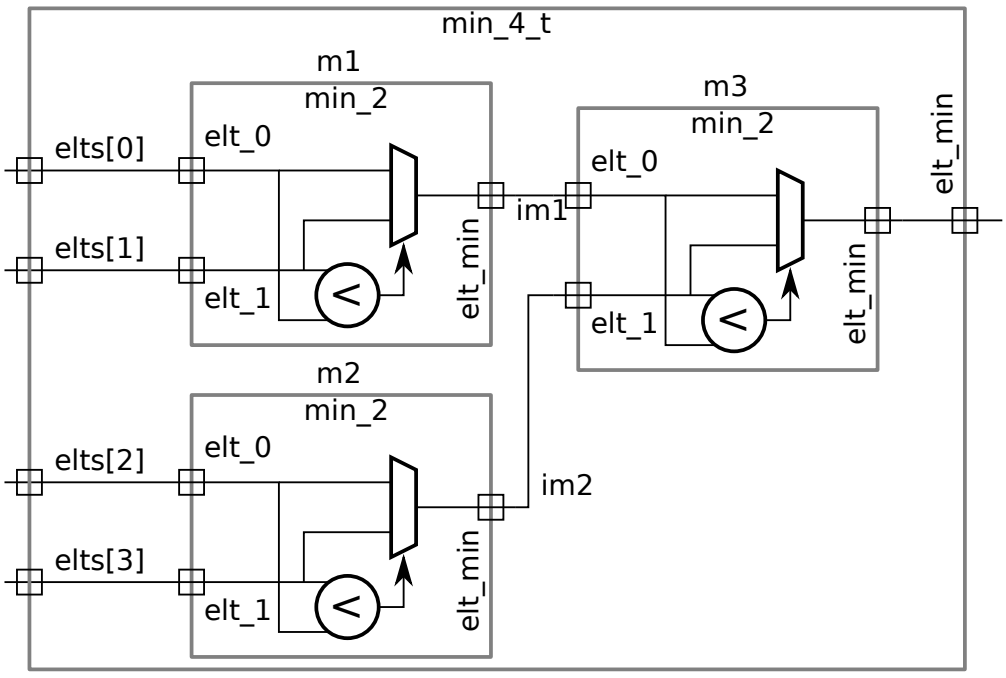
Problem 2: Appearing below are two variations on a `min_4` module that finds the minimum of four unsigned integers. Both of these modules instantiate the following `min_2` module.

```
module min_2
  #( int elt_bits = 4 )
   ( output [elt_bits-1:0] elt_min,
     input [elt_bits-1:0] elt_0,
     input [elt_bits-1:0] elt_1 );
   assign                 elt_min = elt_0 < elt_1 ? elt_0 : elt_1;
endmodule
```

(*a*) Draw a diagram of the hardware that will be synthesized for the `min_4_t` module below. Your diagram should include two-input multiplexors and a comparison module. To get an idea of what to draw, see the EE 3755 Homework solution mentioned at the top of this assignment.

```
module min_4_t
  #( int elt_bits = 4 )
   ( output [elt_bits-1:0] elt_min,
     input [elt_bits-1:0] elts [4] );

   wire [elt_bits-1:0]    im1, im2;
   min_2 #(elt_bits) m1( im1, elts[0], elts[1] );
   min_2 #(elt_bits) m2( im2, elts[2], elts[3] );
   min_2 #(elt_bits) m3( elt_min, im1, im2 );
endmodule
```

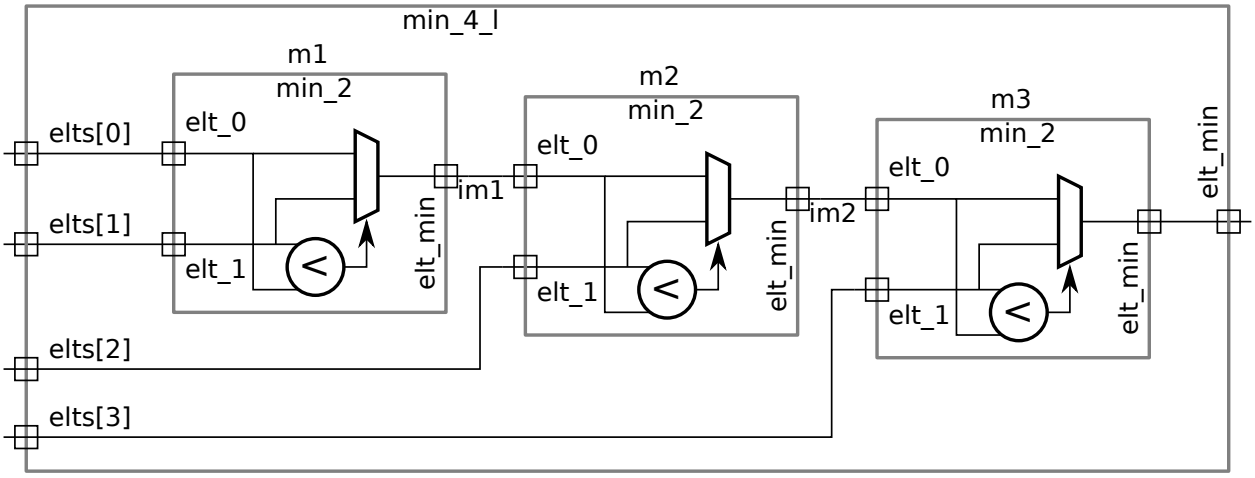
Solution appears below.



(*b*) Draw a diagram of the hardware that will be synthesized for the `min_4_l` module below. Your diagram should include two-input multiplexors and a comparison module.

```
module min_4_l
  #( int elt_bits = 4 )
   ( output [elt_bits-1:0] elt_min,
     input [elt_bits-1:0] elts [4] );

   wire [elt_bits-1:0]    im1, im2;
   min_2 #(elt_bits) m1( im1, elts[0], elts[1] );
   min_2 #(elt_bits) m2( im2, im1, elts[2] );
   min_2 #(elt_bits) m3( elt_min, im2, elts[3] );
endmodule
```

Solution appears below.



(*c*) Which of the two modules above would you expect to have lower cost? Which would you expect to be faster? Briefly explain.

The cost of the two modules should be the same. Module `min_4_t` should be faster because the longest path through the module is through two `min_2` modules, whereas in `min_4_l` the longest paths is through three `min_2` modules.

Problem 3: The module `min_4_err` below is correct Verilog, but it won't do what we want.

```
module min_4_err
  #( int elt_bits = 4 )
   ( output [elt_bits-1:0] elt_min,
     input [elt_bits-1:0] elts [4] );

   wire [elt_bits-1:0]    im;
   min_2 #(elt_bits) m1( im, elts[0], elts[1] );
   min_2 #(elt_bits) m2( im, im, elts[2] );
   min_2 #(elt_bits) m3( elt_min, im, elts[3] );

endmodule
```

(*a*) Explain why it's correct Verilog yet provides the incorrect result.

The problem is that the output of `m1` and `m2` are both connected to the same net, `im`. This may lead to conflicts, for example, when `m1` wants to set bit `im[0]` to `1` but `m2` wants to set it to `0`. The simulator will assign an `x` for such cases. Worse, in `m2` an output and an input are connected to the same loop.

(*b*) Look up `uwire` in the SystemVerilog standard and explain how that might help catching such errors.

Unlike a net of type `wire`, a net of type `uwire` can only be driven by one source. See IEEE Std 1800-2012 Section 6.6.2. A net connected in the same way as `im`, above, would result in a Verilog compiler error.

Problem 4: Appearing below is yet another variation on `min_4`, this one attempting to take advantage of a special case by using generate statements. The module is correctly using generate statements to handle a special case. Do you think the synthesized hardware will be less expensive for the special case *beyond the reduction in cost for using fewer bits*. Hint: Think about what the comparison unit and mux would look like with 1-bit inputs and how such logic can be optimized.

Note: In the original assignment this problem had a typo, which made the Verilog illegal. Further, the phrase above starting "beyond the reduction" was not in the original question, making it difficult to see what was really being asked. The answer below is for the corrected question.

```
module min_4_special1
  #( int elt_bits = 4 )
   ( output [elt_bits-1:0] elt_min,
     input [elt_bits-1:0] elts [4] );

   if ( elt_bits == 1 ) begin

      assign elt_min = elts[0] && elts[1] && elts[2] && elts[3];

   end else begin

      wire [elt_bits-1:0]    im1, im2;

      min_2 #(elt_bits) m1( im1, elts[0], elts[1] );
      min_2 #(elt_bits) m2( im2, im1, elts[2] );
      min_2 #(elt_bits) m3( elt_min, im2, elts[3] );

   end

endmodule
```

The special case is, of course, and AND gate and we expect that the synthesis program can easily handle those. When `elt_bits` is greater than one the synthesis program sees a linear connection of `min_2` modules

Problem 5: The module below handles another special case, in this case the case where the first element is zero.

```
module min_4_special2
  #( int elt_bits = 4 )
   ( output [elt_bits-1:0] elt_min,
     input [elt_bits-1:0] elts [4] );

   wire [elt_bits-1:0]    im1, im2;

   if ( elts[0] == 0 )
     assign elt_min = 0;
   else begin
      min_2 #(elt_bits) m1( im1, elts[0], elts[1] );
      min_2 #(elt_bits) m2( im2, im1, elts[2] );
      min_2 #(elt_bits) m3( elt_min, im2, elts[3] );
   end
endmodule
```

(*a*) Explain why the module is illegal Verilog.

The `if` statement, testing `elts[0]`, is not in procedural code (for example, in an `initial` or `always`), and so it will be interpreted as a generate statement. Generate statements can only access elaborate-time constants, such as parameters and variables declared `genvar`. A module input port, such as `elts`, is definitely not such a constant and so there is an error.

(*b*) Explain why what it's trying to do would be unlikely to help within a larger design. *Hint: Think about critical path.*

Suppose that the delay through `min_4_special2` when `elts[0]==0` is 1 ns and is 3 ns in other cases. Suppose that the output of `min_4_special2` is connected to logic that has another 5 ns of delay. Setting a clock period to $1 + 5 = 6$ ns would result in errors when the special case was not present and setting it to $3 + 5 = 8$ ns would make the special-case hardware unnecessary.

It's not impossible to take advantage of the special case. To do so external logic would need to detect it (an output indicating the special case could be added to `min_4_special2`) and there would have to be some advantage for the special case. One possibility is that for the special case results from the external logic would be captured in one cycle, otherwise it would take two cycles.

LSU EE 4755 Homework 2 Solution Due: 16 September 2015

The Verilog part of the solution to this assignment can be found in `/home/faculty/koppel/pub/ee4755/hw/2015f/hw02/hw02/hw02-sol.v` and a syntax-highlighted version can be found at `http://www.ece.lsu.edu/koppel/v/2015/hw02-sol.v.html`.

Problem 0: Follow the instructions for account setup and homework workflow on the course procedures page, `http://www.ece.lsu.edu/koppel/v/proc.html`. Run the testbench on the unmodified file. There should be errors on all but the `min_4` (Four-element) module. Try modifying `min_4` so that it simulates but produces the wrong answer. Re-run the simulator and verify that it's broken. Then fix it.

Note: There are no points for this problem.

Problem 1: Module `min_n` has an `elt_bits`-bit output `elt_min` and an `elt_count`-element array of `elt_bits`-bit elements, `elts`. Complete `min_n` so that `elt_min` is set to the minimum of the elements in `elts`, interpreting the elements as unsigned integers. Do so using a linear connection of `min_2` modules instantiated with a `genvar` loop. (A linear connection means that the output of instance i is connected to the input of instance $i+1$.)

Verify correct functioning using the testbench.

Solution appears below.

```
module min_n
  #( int elt_bits = 4,
     int elt_count = 8 )
   ( output uwire [elt_bits-1:0] elt_min,
     input uwire [elt_bits-1:0] elts [ elt_count ] );

   /// SOLUTION

   // Declare wires to interconnect the instances of min_2 instantiated
   // in the genvar loop.
   //
   uwire [elt_bits-1:0] im[elt_count:0];  // im: Inter-Module
   assign               im[0] = elts[0];

   // Instantiate elt_count-1 min_2 modules. The inputs of the first
   // module (i=1) connect to elt[0] and elt[1]. Subsequent modules
   // connect to an elt and the module instantiated in the previous
   // iteration.
   //
   for ( genvar i = 1; i < elt_count; i++ )
     min_2 #(elt_bits) m( im[i], elts[i], im[i-1] );

   // Connect the output of the last instance to the module output.
   //
   assign elt_min = im[elt_count-1];

endmodule
```

Problem 2: Module `min_t` is to have the same functionality as `min_n`. Complete `min_t` so that it recursively instantiates itself down to some minimum size. The actual comparison should be done by a `min_2` module.

Verify correct functioning using the testbench.

Solution appears below. In this solution recursion ends when `elt_count` is 1, in which case the module output, `elts_min` is connected directly to the module input, `elts[0]`. Otherwise two smaller `min_t` modules are instantiated.

```
module min_t
  #( int elt_bits = 4,
     int elt_count = 8 )
   ( output uwire [elt_bits-1:0] elt_min,
     input uwire [elt_bits-1:0] elts [ elt_count-1:0 ] );

   /// SOLUTION

   if ( elt_count == 1 ) begin

      // Recursion ends here with one elt. Of course, it is the
      // minimum. (And the maximum, and the average, and the median.)
      //
      assign elt_min = elts[0];

   end else begin

      // If there are at least two elements instantiate two smaller
      // modules.

      // Compute the number of elements to be handled by each
      // module. (Note that elt_count can be odd, which is why we need
      // a separate elt_hi and elt_lo.)
      //
      localparam int elt_hi = elt_count / 2;
      localparam int elt_lo = elt_count - elt_hi;

      // Wires for interconnection of modules.
      uwire [elt_bits-1:0] minl, minh;

      // Recursively declare two modules.
      //
      min_t #(elt_bits,elt_hi) mhi(minl,elts[elt_count-1:elt_lo]);
      min_t #(elt_bits,elt_lo) mlo(minh,elts[elt_lo-1:0]);

      // Combine the output of the two modules above.
      //
      min_2 #(elt_bits) m2(elt_min,minl,minh);

   end

endmodule
```

Problem 3: By default the synthesis script will synthesize each module for two array sizes, four

elements and eight elements.

(*a*) Run the synthesis script unmodified. Use the command `rc -files syn.tcl`. Explain the differences in performance between the different modules.

The output of the synthesis script appears below.

We should expect the cost and performance of `min_n` and `min_b` to be about the since they should synthesize to the same hardware. That can be seen by comparing the `if` statement in `min_b` to the `assign` in `min_2`: both will synthesize to a multiplexor. The behavioral `for` loop in `min_b` and the generate loop in `min_n` should interconnect those multiplexors in the same way. From the table below we see that the synthesis program output is consistent with our expectations.

We should expect the cost of `min_n` and `min_t` to be about the same since they have the same number of comparison units, they are just connected in a different order. But we should expect `min_t` to be faster since the critical path is through $\log_2 n$ `min_2` modules. The delay numbers match our expectations for the eight-element version, but at four elements the linear versions are faster. One reason for this might be that for some reason, the synthesis program is using a higher-cost comparison unit in the linear versions, adding to their cost and improving their performance. In the four-element versions that added performance puts them ahead of the tree version. But for the eight-input versions the tree version is clearly faster.

Possible Test Question: Estimate the critical path in the tree and linear versions of the `min` units.

The second table below shows the synthesis of the modules at a much higher delay target so that the synthesis program will be optimizing primarily for area. In this case the both the cost and performance differences between the tree and linear versions meet our expectations.

Module Name	Area	Delay Actual	Delay Target
min_t_elt_bits4_elt_count4	8592	1416	100
min_b_elt_bits4_elt_count4	14360	1367	100
min_n_elt_bits4_elt_count4	14360	1367	100
min_t_elt_bits4_elt_count8	25536	1935	100
min_b_elt_bits4_elt_count8	29460	3712	100
min_n_elt_bits4_elt_count8	29460	3712	100

Module Name	Area	Delay Actual	Delay Target
min_t_elt_bits4_elt_count4	5180	2413	50000
min_b_elt_bits4_elt_count4	5152	3280	50000
min_n_elt_bits4_elt_count4	5152	3280	50000
min_t_elt_bits4_elt_count8	11784	3609	50000
min_b_elt_bits4_elt_count8	12176	7796	50000
min_n_elt_bits4_elt_count8	12176	7796	50000

(*b*) Modify and re-run the synthesis script so that it synthesizes the modules with `elt_bits` set to 1.

The synthesis program should do a better job on the behavioral and linear models *in comparison to the tree model*. Why do you think that is? *Hint: The 1-bit minimum module is equivalent to another common logic component that the synthesis program can handle well. Note: the phrase about the tree model was not in the original assignment.*

In the table below we see that with a 1-bit element size all three modules have identical cost and performance.

With a one-bit element size the circuit acts as an AND gate, and this is something the synthesis program can figure out. Since the synthesis program sees that `min_n` and `min_b` are performing AND operations it can apply the same kind of tree reduction technique that we incorporated by hand in `min_t`, and so all modules are the same.

Note that the key insight here is that in the general case the synthesis program could not figure out that the minimum operation is associative, and so it could not apply a tree reduction. But with the element size set to 1, it converted minimum to AND, which it did recognize as associative.

Module Name	Area	Delay Actual	Delay Target
min_t_elt_bits1_elt_count4	288	155	100
min_b_elt_bits1_elt_count4	288	155	100
min_n_elt_bits1_elt_count4	288	155	100
min_t_elt_bits1_elt_count8	912	292	100
min_b_elt_bits1_elt_count8	912	292	100
min_n_elt_bits1_elt_count8	912	292	100

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2015 Homework 2 -- SOLUTION
//

 /// Assignment http://www.ece.lsu.edu/koppel/v/2015/hw02.pdf
 /// Solution http://www.ece.lsu.edu/koppel/v/2015/hw02_sol.pdf

 /// Instructions:
  //
  // (1) Find the undergraduate workstation laboratory, room 126 EE
  //     Building.
  //
  // (2) Locate your account.  If you did not get an account please
  //     E-mail: koppel@ece.lsu.edu
  //
  // (3) Log in to a Linux workstation.
  //     The account should start up with a WIMP interface (windows, icons,
  //     mouse, pull-down menus)  ( :-) ) but one or two things need
  //     to be done from a command-line shell.  If you need to brush up
  //     on Unix commands follow http://www.ece.lsu.edu/koppel/v/4ltrwrd/.
  //
  // (4) If you haven't already, follow the account setup instructions here:
  //     http://www.ece.lsu.edu/koppel/v/proc.html
  //
  // (5) Copy this assignment, local path name
  //     /home/faculty/koppel/pub/ee4755/hw/2015f/hw02
  //     to a directory ~/hw02 in your class account. (~ is your home
  //     directory.) Use this file for your solution.
  //
  // (6) Find the problems in this file and solve them.
  //
  //     Your entire solution should be in this file.
  //
  //     Do not change module names.
  //
  // (7) Your solution will automatically be copied from your account by
  //     the TA-bot.


 /// Additional Resources
  //
  // Verilog Documentation
  //    The Verilog Standard
  //      http://standards.ieee.org/getieee/1800/download/1800-2012.pdf
  //    Introductory Treatment (Warning: Does not include SystemVerilog)
  //      Brown & Vranesic, Fundamentals of Digital Logic with Verilog, 3rd Ed.
  //
  // Account Setup and Emacs (Text Editor) Instructions
  //      http://www.ece.lsu.edu/koppel/v/proc.html
  //      To learn Emacs look for Emacs tutorial.
  //
  // Unix Help
  //      http://www.ece.lsu.edu/koppel/v/4ltrwrd/


//  `default_nettype none

////////////////////////////////////////////////////////////////////////////////
/// Problem 0
//
 /// Minimum Modules
//
```

```
//   Look over the code below.
//   There is nothing to turn in for this problem.
//


 /// Behavioral elt_count-input Minimum Module
//
module min_b
  #( int elt_bits = 4,
     int elt_count = 8 )
   ( output logic [elt_bits-1:0] elt_min,
     input uwire [elt_bits-1:0] elts[elt_count] );

   always @* begin

      elt_min = elts[0];

      for ( int i=1; i<elt_count; i++ )
        if ( elts[i] < elt_min ) elt_min = elts[i];

   end

endmodule


 /// Implicit Structural 2-Input Minimum Module
//
module min_2
  #( int elt_bits = 4 )
   ( output uwire [elt_bits-1:0] elt_min,
     input uwire [elt_bits-1:0] elt_0,
     input uwire [elt_bits-1:0] elt_1 );

   assign                elt_min = elt_0 < elt_1 ? elt_0 : elt_1;

endmodule

 /// Explicit Structural 4-Input Minimum Module
//
module min_4
  #( int elt_bits = 4 )
   ( output uwire [elt_bits-1:0] elt_min,
     input uwire [elt_bits-1:0] elts [4] );

   uwire [elt_bits-1:0]    im1, im2;
   min_2 #(elt_bits) m1( im1, elts[0], elts[1] );
   min_2 #(elt_bits) m2( im2, elts[2], elts[3] );
   min_2 #(elt_bits) m3( elt_min, im1, im2 );

endmodule

////////////////////////////////////////////////////////////////////////////////
///  Problem 1 -- SOLUTION
//
 /// Linear Generate minimum module.
//
//   Complete the module.
//
//     [✔] Use a generate loop.
//     [✔] The code must be synthesizable.
//     [✔] Make sure that the testbench does not report errors.

module min_n
  #( int elt_bits = 4,
```

```
    int elt_count = 8 )
  ( output uwire [elt_bits-1:0] elt_min,
    input uwire [elt_bits-1:0] elts [ elt_count ] );

  /// SOLUTION

  // Declare wires to interconnect the instances of min_2 instantiated
  // in the genvar loop.
  //
  uwire [elt_bits-1:0] im[elt_count:0];  // im: Inter-Module
  assign              im[0] = elts[0];

  // Instantiate elt_count-1 min_2 modules. The inputs of the first
  // module (i=1) connect to elt[0] and elt[1]. Subsequent modules
  // connect to an elt and the module instantiated in the previous
  // iteration.
  //
  for ( genvar i = 1; i < elt_count; i++ )
    min_2 #(elt_bits) m(im[i], elts[i], im[i-1] );

  // Connect the output of the last instance to the module output.
  //
  assign        elt_min = im[elt_count-1];

endmodule


////////////////////////////////////////////////////////////////////////////
///  Problem 2
//
 /// Tree Generate minimum module.
//
//   Complete the module.
//
//     [✔] Use recursion: the module should instantiate itself or a min_2.
//     [✔] The code must be synthesizable.
//     [✔] Make sure that the testbench does not report errors.


module min_t
  #( int elt_bits = 4,
     int elt_count = 8 )
   ( output uwire [elt_bits-1:0] elt_min,
     input [elt_bits-1:0] elts [ elt_count-1:0 ] );

  /// SOLUTION

  if ( elt_count == 1 ) begin

     // Recursion ends here with one elt. Of course, it is the
     // minimum. (And the maximum, and the average, and the median.)
     //
     assign elt_min = elts[0];

  end else begin

     // If there are at least two elements instantiate two smaller
     // modules.

     // Compute the number of elements to be handled by each
     // module. (Note that elt_count can be odd, which is why we need
     // a separate elt_hi and elt_lo.)
     //
     localparam int elt_hi = elt_count / 2;
```

```
      localparam int elt_lo = elt_count - elt_hi;

      // Wires for interconnection of modules.
      uwire [elt_bits-1:0] minl, minh;

      // Recursively declare two modules.
      //
      min_t #(elt_bits,elt_hi) mhi(minl,elts[elt_count-1:elt_lo]);
      min_t #(elt_bits,elt_lo) mlo(minh,elts[elt_lo-1:0]);

      // Combine the output of the two modules above.
      //
      min_2 #(elt_bits) m2(elt_min,minl,minh);

   end

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Testbench Code
//
//  The code below instantiates some of the modules above,
//  provides test inputs, and verifies the outputs.
//
//  The testbench may be modified to facilitate your solution. Of
//  course, the removal of tests which your module fails is not a
//  method of fixing a broken module. (The idea is to put in tests
//  which make it easier to determine what the problem is, for
//  example, test inputs that are all 0's or all 1's.)

// cadence translate_off

module testbench;

   testbench_sz #(1,4) t0();
   testbench_sz #(4,4) t1();
   testbench_sz #(8,32) t2();
   testbench_sz #(7,17) t3();

endmodule

module testbench_sz
  #( int elt_bits = 8,
     int elt_count = 80 );

   localparam int mut_cnt_max = 5;

   logic [elt_bits-1:0] elts[elt_count];

   uwire [elt_bits-1:0]  elt_m[mut_cnt_max];
   struct { int err_cnt = 0; int idx; } md[string];

   min_b #(elt_bits,elt_count) m0(elt_m[0],elts);
   min_n #(elt_bits,elt_count) m1(elt_m[1],elts);
   if ( elt_count == 4 )
     min_4 #(elt_bits) m2(elt_m[2],elts);

   min_t #(elt_bits,elt_count) m3(elt_m[3],elts);

   localparam int num_tests = 10000;

   initial begin
```

```
      md["Linear Generate"].idx = 1;
      md["Tree Generate"].idx = 3;
      if ( elt_count == 4 )
        md["Four-Element"].idx = 2;

      for ( int i=0; i<num_tests; i++ ) begin

         for ( int j=0; j<elt_count; j++ ) elts[j] = $random();

         #1;

         foreach ( md[mut] ) begin

            if ( elt_m[0] !== elt_m[md[mut].idx] ) begin

               md[mut].err_cnt++;
               if ( md[mut].err_cnt < 5 )
                 $write("Error test %0d for %s, 0x%x != 0x%x (correct)\n",
                          i, mut, elt_m[md[mut].idx], elt_m[0] );
            end

         end

      end

      foreach ( md[mut] )
         $write("Tests completed for %s at %0d x %0d, error count %0d\n",
                  mut, elt_bits, elt_count, md[mut].err_cnt );

   end

endmodule

// cadence translate_on
```

LSU EE 4755 Homework 3 Solution Due: 7 October 2015

Problem 1: Solve EE 4755 Fall 2014 Midterm Exam Problem 4 and Problem 5. The solutions are available, but please make an honest effort to solve them on your own.

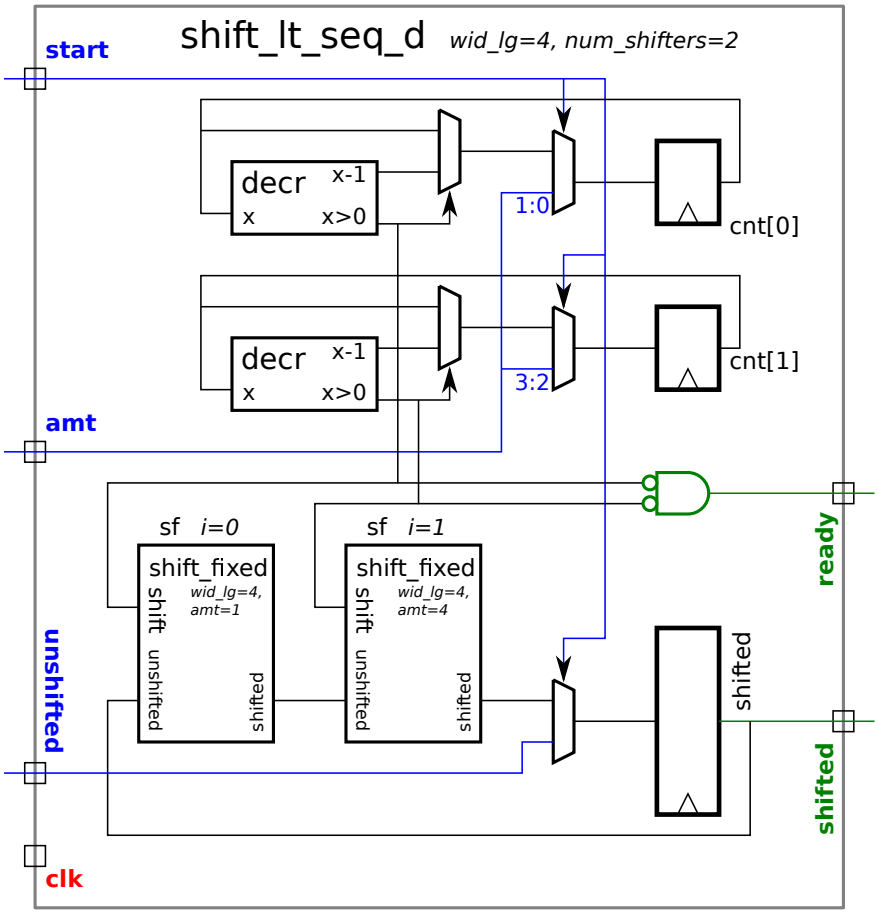
See the posted solutions at `http://www.ece.lsu.edu/koppel/v/2014/mt_sol.pdf`.

Problem 2: The homework Verilog file, `hw04.v` contains two versions of the sequential shifter used in class, those modules are also reproduced below. Module `shift_lt_seq_d_live`, is based on the version written during class and module `shift_lt_seq_d` is the one prepared in advance. Though both work correctly their timing is not identical.

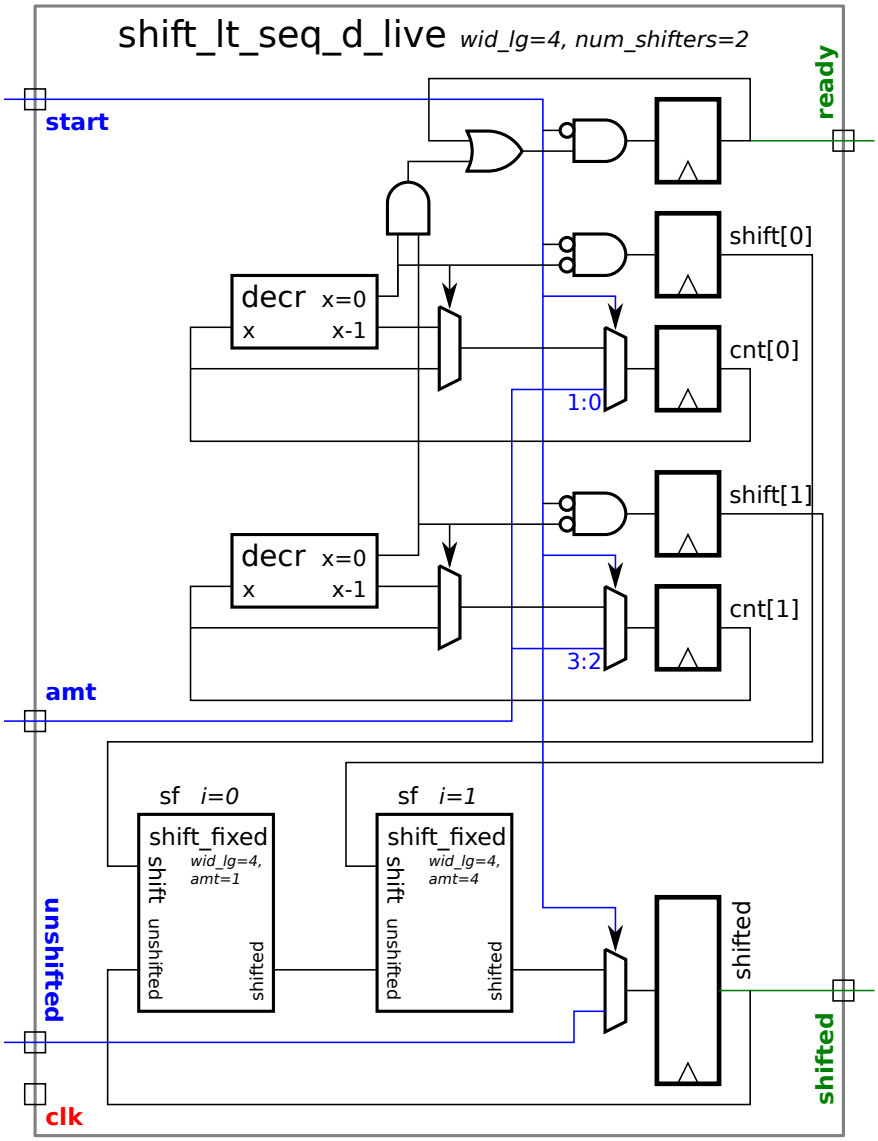
(*a*) Show the hardware that might be synthesized for each module using the default parameters. Include reasonable optimizations, the initially inferred hardware can be omitted. This should be a human-to-human diagram, don't show the output of a synthesis program.

Note: In the original assignment the parameters for the `shift_lt_seq_d_live` *module were not set as intended, that has been corrected in this version of the homework assignment. Both solutions appear below, they are referred to as the* original *and* intended *module. In the intended assignment (this one) both modules have the same parameters, in the original assignment the live module had just one shifter and could shift more bits.*

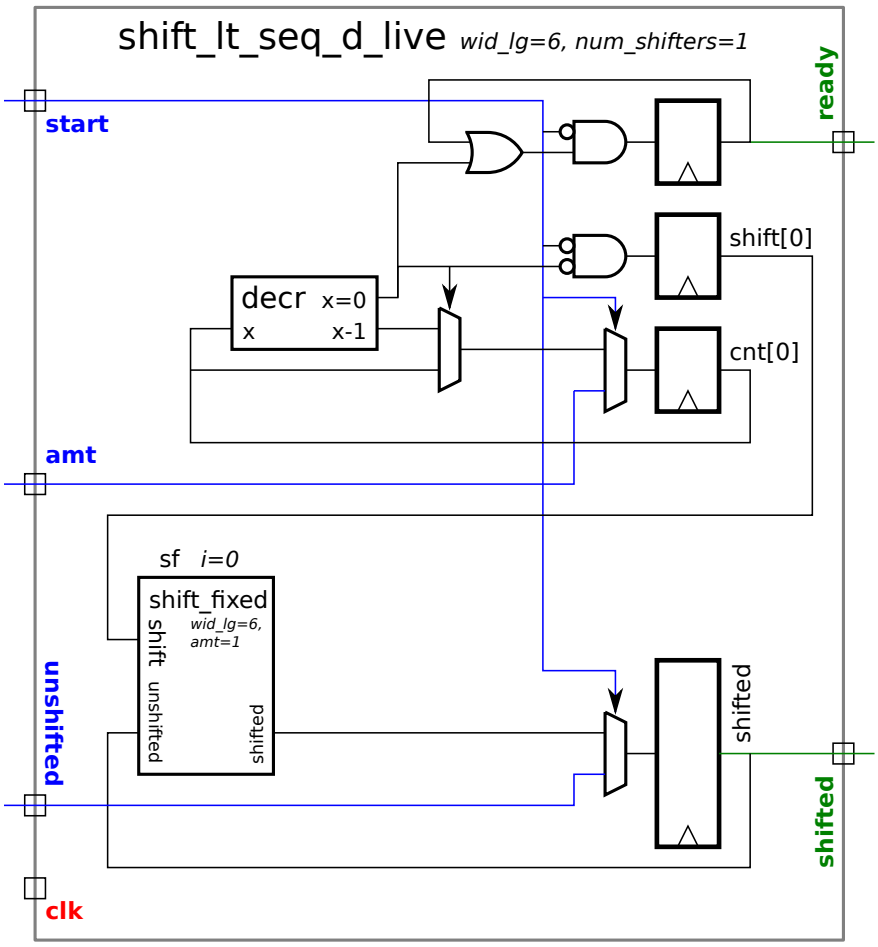
The hardware appears below. In **shift_lt_seq_d_live** the initially inferred multiplexors at the inputs to the **ready** and **shift** registers have been replaced by logic gates. The logic computing the next state of **ready** includes the old value of **ready**. The old value of **ready** isn't really needed, but it's shown because it is probably what the synthesis program would have included.



The intended live module appears below:



The original live module appears below:



(*b*) The two modules differ in their timing. Using your hardware diagrams explain any differences in:

- The register-to-register delay within the module.
- How far in advance of the positive edge module inputs must become stable.
- How long after the positive edge module outputs will be available.

As with the previous part, this should be done by hand though synthesis tools can be used to help solve the problem.

An answer might look like this: *"For register-to-register delay Module A is slower because its critical path has two multipliers, whereas in module B the two multiplications are split between cycles and so at most one multiplier is on the critical path. In module A inputs connect directly to a divider, and so they must arrive long before the positive edge, whereas in module B inputs can arrive just before the positive edge because ...."* Of course, this question does not have a module A or B, nor does it really have multipliers and dividers.

The following timing will be assumed when comparing the modules. Multiplexor delay is two gate delays from either the select or data inputs. For a two-bit decrementor the `x=0`, `x>0`, and `x-1` outputs are all 1 gate delay (draw a truth table). A six-bit decrementor is assumed to take two gate delays to compute `x=0` and 6 gate delays to compute `x-1`. Since it's essentially a multiplexor the `shift_fixed` modules take two gate delays regardless of the shift amount.

An important difference between the live and prepared module, is that the in the live module the **shift** input to **shift_fixed** comes from a register output, and so it will be available at the beginning of a the clock cycle. In the prepared module the **shift** input is generated by checking if a portion of **cnt** is zero, the check adds a small delay. Though this may sound like a small advantage for the live module, but it may not be because it doesn't use the **shift** signal until the next clock cycle and so it takes one clock cycle longer to perform the shift. If **wid_lg/num_shifters** is large than the extra clock cycle will be a small fraction of the total time and so the live module would be better. If the ratio is small the extra clock cycle will make things slower.

For the assigned problem, in which **shift_lt_seq_d_live** has 1 shifter, the register-to-register critical path in the live module is 10 gate delays, assuming 6 gate delays for the 6-bit subtract. The prepared module, **shift_lt_seq_d**, module has a critical path of 7 delays. Thus, the live module can have a higher clock frequency—that's the good news—but it will take $\frac{2^6}{2^{4/2}-1} = 21.33$ times as many cycles to perform the largest shift.

A concise answer to the assigned problem might be: the register-to-register delay in the live module is much longer because it must decrement a much larger number, six versus two bits. This overcomes any benefit of having one shifter, versus two in the prepared module.

In the intended problem the live module has the same parameters as the prepared module, including two shifters. In that case the critical path is 6 gate delays, 1 gate delay faster than the prepared module. But because it takes one cycle longer the benefit in clock frequency would not be large enough to overcome the disadvantage of requiring one more clock cycle, at least not for the default parameters.

The two modules have equivalent input setup times, two gate delays. So for both, the inputs can arrive near the end of the clock cycle.

In the live module the outputs are available at the beginning of the clock cycle. In the prepared module the **ready** signal is generated using an AND gate connected to the decrementors. Based on the analysis above, the prepared module's ready output is not available until two gate delays after the clock edge.

Modules on next page.

```
module shift_lt_seq_d_live
  #( int wid_lg = 4,        // In original assignment, 6
     int num_shifters = 2,  // In original assignment, 1.
     int wid = 1 << wid_lg )
   ( output logic [wid-1:0] shifted,
     output logic ready,
     input [wid-1:0] unshifted,
     input [wid_lg-1:0] amt,
     input start,
     input clk );

   localparam int bits_per_seg = wid_lg / num_shifters;

   logic [num_shifters-1:0] shift;
   wire [wid-1:0] shin[num_shifters-1:-1];
   assign shin[-1] = shifted;

   for ( genvar i=0; i<num_shifters; i++ ) begin
      localparam int fs_amt = 2 ** ( i * bits_per_seg );
      shift_fixed #( wid_lg, fs_amt ) sf( shin[i], shin[i-1], shift[i] );
   end

   logic [num_shifters-1:0][bits_per_seg-1:0] cnt;

   always_ff @( posedge clk ) begin

      if ( start == 1 ) begin
         ready = 0;
         cnt = amt;
         shift = 0;
         shifted = unshifted;
      end else begin
         if ( cnt == 0 ) ready = 1;
         for ( int i=0; i<num_shifters; i++ ) begin
            shift[i] = cnt[i] > 0;
            if ( cnt[i] != 0 ) cnt[i]--;
         end
         shifted = shin[num_shifters-1];
      end

   end

endmodule
```

Another module on next page.

```
module shift_lt_seq_d
  #( int wid_lg = 4,
     int num_shifters = 2,
     int wid = 1 << wid_lg )
   ( output logic [wid-1:0] shifted,
     output wire ready,
     input [wid-1:0] unshifted,
     input [wid_lg-1:0] amt,
     input start,
     input clk );

   localparam int cnt_bits = ( wid_lg + num_shifters - 1 ) / num_shifters;
   logic [num_shifters-1:0][cnt_bits-1:0] cnt;
   wire [wid-1:0] inter_sh[num_shifters-1:-1];
   assign inter_sh[-1] = shifted;

   for ( genvar i = 0; i < num_shifters; i++ ) begin
      localparam int shift_amt = 1 << i * cnt_bits;
      wire       shift = cnt[i] != 0;
      shift_fixed #(wid_lg,shift_amt) sf( inter_sh[i], inter_sh[i-1], shift );
   end

   always_ff @( posedge clk )

      if ( start == 1 ) begin
         shifted = unshifted;
         cnt = amt;
      end else if ( cnt > 0 ) begin
         shifted = inter_sh[num_shifters-1];
         for ( int i=0; i<num_shifters; i++ ) if ( cnt[i] ) cnt[i]--;
      end

   assign ready = cnt == 0;

endmodule
```

LSU EE 4755 Homework 4 Solution Due: 12 October 2015

Problem 0: Follow the instructions for account setup and homework workflow on the course procedures page, `http://www.ece.lsu.edu/koppel/v/proc.html`. Run the testbench on the unmodified file. There should be errors on the `shift_lt_seq_d_sol` module, but the others should run correctly. Run the Note: There are no points for this problem.

Problem 1: The homework Verilog file, `hw04.v`, contains a module `shift_lt_seq_d_sol` which is based on `shift_lt_seq_d`. It contains an `always_ff` block that assigns the same variables that are assigned in `shift_lt_seq_d`, however it assigns them from variables of the same name with `next_` prefixed:

```
always_ff @( posedge clk ) begin
   ready = next_ready;
   shifted = next_shifted;
   shift = next_shift;
   cnt = next_cnt;
end
```

Add code so that these `next_` objects will be assigned values from combinational logic, and so that the resulting module describes the same hardware as `shift_lt_seq_d`. A hand-drawn diagram of synthesized hardware should be identical, though it's possible that there will be small differences in the actual output of a synthesis program.

The added code can be implicit structural or behavioral, but it must synthesize to combinational logic.

The simplest approach is to start with the `always_ff` block from module `shift_lt_seq_d`. Change the always type to `always_comb` and rename some of the objects that are to synthesize to registers, namely `ready`, `shifted`, `shift`, and `cnt`.

If an assignment is made to any of these in the `always_comb` block, the assignment must be changed to write the `next_` version. For example change `cnt=amt;` to `next_cnt=amt;`. The right-hand side of an assignment should only use the `next_` version of a variable if it was assigned earlier in the block. For example, `next_shift` in the excerpt from the solution below:

```
next_shift[i] = cnt[i] > 0;
next_cnt[i] = next_shift[i] ? cnt[i] - 1 : cnt[i];
```

The code also has to be modified so that each of the `next_` variables is assigned at least once no matter what path is taken through the `always_comb` block. That is, they must be assigned for every possible outcome of the `if` statements. That's why there is no `if` statement in the assignment to `next_cnt` above. (That is, the following would be wrong: `if(next_shift[i])next_cnt[i]=cnt[i]-1`). (If a variable is not always assigned then its value will come from the output of a latch, rather than from combinational logic.)

The solution uses both continuous assign statements and an `always_comb` block. The complete solution appears below:

```
module shift_lt_seq_d_sol
  #( int wid_lg = 4, int num_shifters = 2, int wid = 1 << wid_lg )
   ( output logic [wid-1:0] shifted,       output logic ready,
     input [wid-1:0] unshifted,            input [wid_lg-1:0] amt,
     input start,                          input clk );

   logic [num_shifters-1:0] shift;
```

```
   wire [wid-1:0]           shin[num_shifters-1:-1];
   localparam int bits_per_seg = wid_lg / num_shifters;
   for ( genvar i=0; i<num_shifters; i++ ) begin
      localparam int fs_amt = 2 ** ( i * bits_per_seg );
      shift_fixed #( wid_lg, fs_amt ) sf( shin[i], shin[i-1], shift[i] );
   end

   assign shin[-1] = shifted;

   logic [num_shifters-1:0][bits_per_seg-1:0] cnt;
   logic [wid-1:0] next_shifted;
   logic next_ready;
   logic [num_shifters-1:0] next_shift;
   logic [num_shifters-1:0][bits_per_seg-1:0] next_cnt;

   always_comb begin

      if ( start == 1 ) begin

         next_cnt = amt;
         next_shift = 0;

      end else begin

         for ( int i=0; i<num_shifters; i++ ) begin
            next_shift[i] = cnt[i] > 0;
            // Note that next_cnt is always assigned, this avoids latches.
            next_cnt[i] = next_shift[i] ? cnt[i] - 1 : cnt[i];
         end
      end
   end

   // Use a continuous assignment for next_ready and next_shifted.
   assign next_ready = start ? 0 : cnt == 0 ? 1 : ready;
   assign next_shifted = start ? unshifted : shin[num_shifters-1];

   always_ff @( posedge clk ) begin
      shifted = next_shifted;
      ready = next_ready;
      shift = next_shift;
      cnt = next_cnt;
   end

endmodule
```

Problem 2: Module `shift_lt_seq_d_live` takes one more cycle to produce a result than module `shift_lt_seq_d`. Module `shift_lt_seq_d_p2` initially is identical to `shift_lt_seq_d_live`.

(*a*) Modify `shift_lt_seq_d_p2` so that it uses one less cycle to produce a result without changing the number of shifters per stage. There are two possible ways of doing this, performing some work in the same cycle that the `start` signal arrives, or doing work in the cycle when `ready` is set to 1. Either method is fine.

The original module, **shift_lt_seq_d_live**, does not start to shift until the cycle after **start** is set to 1. In the solution the logic generating the **shift** signal is moved so that it operates at every cycle. That was done by moving the **i** loop out of the **if/else** block, the logic generating the **ready** signal was also moved.

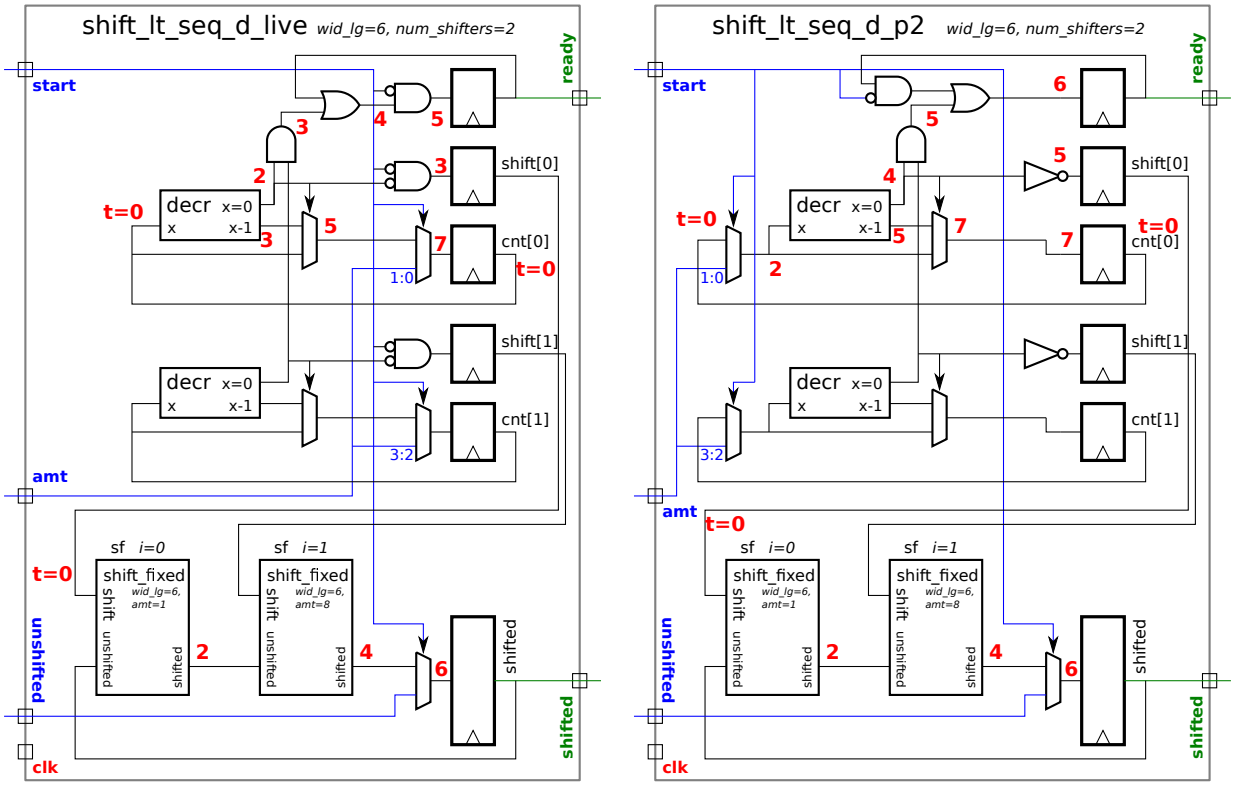
By doing this we are requiring **start** and **amt** to arrive early in the cycle. Before the change they could arrive late in the cycle.

```
module shift_lt_seq_d_p2
  #( int wid_lg = 6, int num_shifters = 1, int wid = 1 << wid_lg )
   ( output logic [wid-1:0] shifted,       output logic ready,
     input [wid-1:0] unshifted,            input [wid_lg-1:0] amt,
     input start,                          input clk );

   localparam int bits_per_seg = wid_lg / num_shifters;

   logic [num_shifters-1:0] shift;
   wire [wid-1:0] shin[num_shifters-1:-1];
   assign shin[-1] = shifted;

   for ( genvar i=0; i<num_shifters; i++ ) begin
      localparam int fs_amt = 2 ** ( i * bits_per_seg );
      shift_fixed #( wid_lg, fs_amt ) sf( shin[i], shin[i-1], shift[i] );
   end

   logic [num_shifters-1:0][bits_per_seg-1:0] cnt;

   always_ff @( posedge clk ) begin

      if ( start == 1 ) begin
         ready = 0;
         cnt = amt;
         shifted = unshifted;
      end else begin
         shifted = shin[num_shifters-1];
      end

      if ( cnt == 0 ) ready = 1;

      for ( int i=0; i<num_shifters; i++ ) begin
         shift[i] = cnt[i] > 0;
         if ( cnt[i] != 0 ) cnt[i]--;
      end

   end

endmodule
```

(*b*) Run **syn.tcl** and compare the cost and performance of your design and **shift_lt_seq_d_live**. Comment on the differences. An answer might start *"The cost was about the same because the same hardware was used..."*.

A table showing area (cost) and timing as reported by the synthesis program appears below. That's followed by a sketch of our guess of the synthesized hardware for each module, along with a timing analysis. These expectations are compared with the output of the synthesis program.

Module Name	Area	Delay Actual	Delay Target
shift_lt_seq_d_live_wid_lg6_num_shifters1	68368	1253	100
shift_lt_seq_d_p2_wid_lg6_num_shifters1	68428	1229	100
shift_lt_seq_d_live_wid_lg6_num_shifters2	77528	1355	100
shift_lt_seq_d_p2_wid_lg6_num_shifters2	78700	1348	100
shift_lt_seq_d_live_wid_lg6_num_shifters3	96648	1527	100
shift_lt_seq_d_p2_wid_lg6_num_shifters3	95820	1539	100
shift_lt_seq_d_live_wid_lg6_num_shifters6	143412	2002	100
shift_lt_seq_d_p2_wid_lg6_num_shifters6	142380	2007	100



To determine the expected area and timing differences between the two modules examine the sketches of the expected synthesized hardware for the two modules, which appears above. The change that enables us to save a cycle is moving the mux that selects a new value of **amt** from the input of **cnt** to the input of the decrement unit. That lets the shifter get started one cycle earlier.

Notice that by moving the hardware to compute **cnt** and **shift** out of the loop we are simplifying the logic at the input to those registers because they no longer have to check **start**. For this reason we would expect the cost to be slightly lower. The costs reported by the synthesis program are close and show no consistent pattern.

The sketches of the expected hardware include a simple timing analysis. The timing analysis is based on an assumed delay of two units for a mux, $\lceil \lg n \rceil$ units for an n-input gate and a delay of 3 for a 3-bit decrementor.

Based on this analysis the changes in the **p2** module don't affect the path that ends in the **shifted** register, that's the same 6 units in both cases.

Moving the **amt** mux from **cnt** to the decrementer inputs does not change the critical path. The move does delay the **shift** and **ready** signals by one or two units, but since they are not critical it doesn't matter. When **num_shifters** is 1 the path ending at **cnt** remains critical so moving the **mux** doesn't change anything. When **num_shifters** is larger the path ending at **shifted** is critical so moving the mux has no impact.

Based on this analysis we would not expect a change in the clock period. The output of the synthesis program shows only small changes.

The fact that the clock period is about the same is good news for us since one less clock cycle is needed. If the changes increased the clock period we may not actually get higher performance.

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2015 Homework 4
//
 /// SOLUTION

 /// Assignment http://www.ece.lsu.edu/koppel/v/2015/hw04.pdf
 /// Solution discussion http://www.ece.lsu.edu/koppel/v/2015/hw04_sol.pdf

 /// Instructions:
  //
  // (1) Find the undergraduate workstation laboratory, room 126 EE
  //     Building.
  //
  // (2) Locate your account.  If you did not get an account please
  //     E-mail: koppel@ece.lsu.edu
  //
  // (3) Log in to a Linux workstation.
  //     The account should start up with a WIMP interface (windows, icons,
  //     mouse, pull-down menus)  ( :-) ) but one or two things need
  //     to be done from a command-line shell.  If you need to brush up
  //     on Unix commands follow http://www.ece.lsu.edu/koppel/v/4ltrwrd/.
  //
  // (4) If you haven't already, follow the account setup instructions here:
  //     http://www.ece.lsu.edu/koppel/v/proc.html
  //
  // (5) Copy this assignment, local path name
  //     /home/faculty/koppel/pub/ee4755/hw/2015f/hw04
  //     to a directory ~/hw04 in your class account. (~ is your home
  //     directory.) Use this file for your solution.
  //
  // (6) Find the problems in this file and solve them.
  //
  //     Your entire solution should be in this file.
  //
  //     Do not change module names.
  //
  // (7) Your solution will automatically be copied from your account by
  //     the TA-bot.


 /// Additional Resources
  //
  // Verilog Documentation
  //    The Verilog Standard
  //      http://standards.ieee.org/getieee/1800/download/1800-2012.pdf
  //    Introductory Treatment (Warning: Does not include SystemVerilog)
  //      Brown & Vranesic, Fundamentals of Digital Logic with Verilog, 3rd Ed.
  //
  // Account Setup and Emacs (Text Editor) Instructions
  //      http://www.ece.lsu.edu/koppel/v/proc.html
  //      To learn Emacs look for Emacs tutorial.
  //
```

```
  // Unix Help
  //      http://www.ece.lsu.edu/koppel/v/4ltrwrd/



////////////////////////////////////////////////////////////////////////////////
///  Problem 0
//
 ///  Shift Left Modules
//
//   Look over the code below.
//   There is nothing to turn in for this problem.
//

`default_nettype none

module shift_fixed
  #( int wid_lg = 4,
     int amt = 1,
     int wid = 1 << wid_lg )
   ( output uwire [wid-1:0] shifted,
     input uwire [wid-1:0] unshifted,
     input uwire shift );

   assign  shifted = shift ? unshifted << amt : unshifted;

endmodule

module shift_lt_behav
  #( int wid_lg = 4,
     int wid = 1 << wid_lg )
   ( output uwire [wid-1:0] shifted,
     input uwire [wid-1:0] unshifted,
     input uwire [wid_lg-1:0] amt );

   assign shifted = unshifted << amt;

endmodule

module shift_lt_comb
  #( int wid_lg = 4,
     int wid = 1 << wid_lg )
   ( output uwire [wid-1:0] shifted,
     input uwire [wid-1:0] unshifted,
     input uwire [wid_lg-1:0] amt );

   uwire [wid-1:0]       step[wid_lg-1:-1];

   assign step[-1] = unshifted;
   assign shifted = step[wid_lg-1];

   for ( genvar i=0; i<wid_lg; i++ )
     shift_fixed #(wid_lg,1<<i) sf( step[i], step[i-1], amt[i] );
```

```
endmodule

module shift_lt_seq
  #( int wid_lg = 4,
     int wid = 1 << wid_lg )
   ( output logic [wid-1:0] shifted,
     output uwire ready,
     input uwire [wid-1:0] unshifted,
     input uwire [wid_lg-1:0] amt,
     input uwire start,
     input uwire clk );

   logic [wid_lg-1:0]   cnt;

   uwire [wid-1:0]       sf_out;

   shift_fixed #(wid_lg,1) sf( sf_out, shifted, 1'b1 );

   always_ff @( posedge clk ) begin

      if ( start == 1 ) begin

         shifted = unshifted;
         cnt = amt;

      end else if ( cnt > 0 ) begin

         shifted = sf_out;
         cnt--;

      end

   end

   assign ready = cnt == 0;

endmodule



module shift_lt_seq_d
  #( int wid_lg = 4,
     int num_shifters = 2,
     int wid = 1 << wid_lg )
   ( output logic [wid-1:0] shifted,
     output uwire ready,
     input uwire [wid-1:0] unshifted,
     input uwire [wid_lg-1:0] amt,
     input uwire start,
     input uwire clk );

   localparam int cnt_bits = ( wid_lg + num_shifters - 1 ) / num_shifters;
   logic [num_shifters-1:0][cnt_bits-1:0] cnt;
   uwire [wid-1:0] inter_sh[num_shifters-1:-1];
```

```
   assign inter_sh[-1] = shifted;

   for ( genvar i = 0; i < num_shifters; i++ ) begin

      localparam int shift_amt = 1 << i * cnt_bits;
      uwire       shift = cnt[i] != 0;

      shift_fixed #(wid_lg,shift_amt) sf( inter_sh[i], inter_sh[i-1], shift );

   end

   always_ff @( posedge clk )

      if ( start == 1 ) begin

         shifted = unshifted;
         cnt = amt;

      end else if ( cnt > 0 ) begin

         shifted = inter_sh[num_shifters-1];
         for ( int i=0; i<num_shifters; i++ ) if ( cnt[i] ) cnt[i]--;

      end

   assign ready = cnt == 0;

endmodule


////////////////////////////////////////////////////////////////////////////////
///  Problem 1
//
//   Modify shift_lt_seq_d_sol so that it synthesizes to the same
//   hardware as shift_lt_seq_d_live (further below).
//
//     [✔] Be sure that all code that you add synthesizes to
//         combinational logic.
//
//     [✔] Make sure that the module runs correctly.
//


module shift_lt_seq_d_sol
  #( int wid_lg = 4,
     int num_shifters = 2,
     int wid = 1 << wid_lg )
   ( output logic [wid-1:0] shifted,
     output logic ready,
     input uwire [wid-1:0] unshifted,
     input uwire [wid_lg-1:0] amt,
     input uwire start,
     input uwire clk );
```

```
   logic [num_shifters-1:0] shift;

   uwire [wid-1:0]           shin[num_shifters-1:-1];

   localparam int bits_per_seg = wid_lg / num_shifters;

   for ( genvar i=0; i<num_shifters; i++ ) begin

      localparam int fs_amt = 2 ** ( i * bits_per_seg );

      shift_fixed #( wid_lg, fs_amt ) sf( shin[i], shin[i-1], shift[i] );

   end

   assign shin[-1] = shifted;

   logic [num_shifters-1:0][bits_per_seg-1:0] cnt;

   logic [wid-1:0] next_shifted;
   logic next_ready;
   logic [num_shifters-1:0] next_shift;
   logic [num_shifters-1:0][bits_per_seg-1:0] next_cnt;

   /// Problem 1: Modify this module, especially around here.

   /// SOLUTION
   //
   //  Some logic from shift_lt_seq_d has been placed into the
   //  always_comb block and some has been placed in assigns.
   //  It would be equally correct to put all of the logic in
   //  an always_comb block (or blocks) or to put all of the logic
   //  in assign statements.  The deciding factor should be on how
   //  easy it is to read the code.

   always_comb begin

      if ( start == 1 ) begin

         next_cnt = amt;
         next_shift = 0;

      end else begin

         for ( int i=0; i<num_shifters; i++ ) begin
            next_shift[i] = cnt[i] > 0;

            // Note that next_cnt is always assigned, this avoids latches.
            next_cnt[i] = next_shift[i] ? cnt[i] - 1 : cnt[i];
         end

      end

   end
```

```
   // Use a continuous assignment for next_ready and next_shifted.
   assign next_ready = start ? 0 : cnt == 0 ? 1 : ready;
   assign next_shifted = start ? unshifted : shin[num_shifters-1];

   always_ff @( posedge clk ) begin

      shifted = next_shifted;
      ready = next_ready;
      shift = next_shift;
      cnt = next_cnt;

   end

endmodule

module shift_lt_seq_d_live
  #( int wid_lg = 6,
     int num_shifters = 1,
     int wid = 1 << wid_lg )
   ( output logic [wid-1:0] shifted,
     output logic ready,
     input uwire [wid-1:0] unshifted,
     input uwire [wid_lg-1:0] amt,
     input uwire start,
     input uwire clk );

   /// DO NOT modify this module.

   localparam int bits_per_seg = wid_lg / num_shifters;

   logic [num_shifters-1:0] shift;
   uwire [wid-1:0] shin[num_shifters-1:-1];
   assign shin[-1] = shifted;

   for ( genvar i=0; i<num_shifters; i++ ) begin

      localparam int fs_amt = 2 ** ( i * bits_per_seg );

      shift_fixed #( wid_lg, fs_amt ) sf( shin[i], shin[i-1], shift[i] );

   end

   logic [num_shifters-1:0][bits_per_seg-1:0] cnt;

   always_ff @( posedge clk ) begin

      if ( start == 1 ) begin

         ready = 0;
         cnt = amt;
         shift = 0;
         shifted = unshifted;

      end else begin
```

```
         if ( cnt == 0 ) ready = 1;

         for ( int i=0; i<num_shifters; i++ ) begin
            shift[i] = cnt[i] > 0;
            if ( cnt[i] != 0 ) cnt[i]--;
         end

         shifted = shin[num_shifters-1];

      end

   end

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Problem 2
//
//   Modify shift_lt_seq_d_p2 so that it uses one less cycle.
//
//     [✔] Make sure that the module runs correctly.
//     [✔] Don't change the number of shifters per stage.


module shift_lt_seq_d_p2
  #( int wid_lg = 6,
     int num_shifters = 1,
     int wid = 1 << wid_lg )
   ( output logic [wid-1:0] shifted,
     output logic ready,
     input uwire [wid-1:0] unshifted,
     input uwire [wid_lg-1:0] amt,
     input uwire start,
     input uwire clk );

   localparam int bits_per_seg = wid_lg / num_shifters;

   logic [num_shifters-1:0] shift;
   uwire [wid-1:0] shin[num_shifters-1:-1];
   assign shin[-1] = shifted;

   for ( genvar i=0; i<num_shifters; i++ ) begin

      localparam int fs_amt = 2 ** ( i * bits_per_seg );

      shift_fixed #( wid_lg, fs_amt ) sf( shin[i], shin[i-1], shift[i] );

   end

   logic [num_shifters-1:0][bits_per_seg-1:0] cnt;

   always_ff @( posedge clk ) begin
```

```
      if ( start == 1 ) begin

         ready = 0;
         cnt = amt;

         shifted = unshifted;

      end else begin

         shifted = shin[num_shifters-1];

      end

      /// SOLUTION
      //
      //  Set shift and update cnt whether or not start==1.
      //

      if ( cnt == 0 ) ready = 1;

      for ( int i=0; i<num_shifters; i++ ) begin
         shift[i] = cnt[i] > 0;
         if ( cnt[i] != 0 ) cnt[i]--;
      end

   end

endmodule




////////////////////////////////////////////////////////////////////////////////
/// Testbench Code
//
//  The code below instantiates some of the modules above,
//  provides test inputs, and verifies the outputs.
//
//  The testbench may be modified to facilitate your solution. Of
//  course, the removal of tests which your module fails is not a
//  method of fixing a broken module. (The idea is to put in tests
//  which make it easier to determine what the problem is, for
//  example, test inputs that are all 0's or all 1's.)


// cadence translate_off


program reactivate(output uwire clk_reactive, input uwire clk);
   assign clk_reactive = clk;
endprogram
```

```
module testbench;

   localparam int wid_lg = 6;
   localparam int wid = 1 << wid_lg;

   localparam int max_units = 20;

   logic      clk;
   bit        done;
   int cycle;

   uwire [wid-1:0] sout[max_units];
   uwire ready[max_units];
   logic [wid-1:0] sin;
   logic [wid_lg-1:0] amt;
   logic              start;

   typedef struct { int idx; int err_count = 0; bit seq = 0;
                    logic [wid-1:0] sout = 'h111; int cyc_tot = 0; } Info;
   Info pi[string];

   shift_lt_seq_d #(wid_lg,1) my_sld4(sout[4], ready[4], sin, amt, start, clk);
   initial begin
      automatic string m = "Degree 1";
      pi[m].idx = 4; pi[m].seq = 1;
   end

   shift_lt_seq_d #(wid_lg,3) my_sld5(sout[5], ready[5], sin, amt, start, clk);
   initial begin
      automatic string m = "Degree 3";
      pi[m].idx = 5; pi[m].seq = 1;
   end

   shift_lt_seq_d_live #(wid_lg,1) my_sld9(sout[9], ready[9], sin, amt, start, clk);
   initial begin
      automatic string m = "Degree 1 live";
      pi[m].idx = 9; pi[m].seq = 1;
   end

   shift_lt_seq_d_live #(wid_lg,3) my_sld2(sout[2], ready[2], sin, amt, start, clk);
   initial begin
      automatic string m = "Degree 3 live";
      pi[m].idx = 2; pi[m].seq = 1;
   end

   shift_lt_seq_d_sol #(wid_lg,1) my_sld1(sout[1], ready[1], sin, amt, start, clk);
   initial begin
      automatic string m = "Degree 1 sol";
      pi[m].idx = 1; pi[m].seq = 1;
   end

   shift_lt_seq_d_sol #(wid_lg,3) my_sld10(sout[10], ready[10], sin, amt, start, clk);
   initial begin
      automatic string m = "Degree 3 sol";
```

```
      pi[m].idx = 10; pi[m].seq = 1;
   end

   shift_lt_seq_d_p2 #(wid_lg,1) my_sld3(sout[3], ready[3], sin, amt, start, clk);
   initial begin
      automatic string m = "Degree 1 P2";
      pi[m].idx = 3; pi[m].seq = 1;
   end

   shift_lt_seq_d_p2 #(wid_lg,3) my_sld6(sout[6], ready[6], sin, amt, start, clk);
   initial begin
      automatic string m = "Degree 3 P2";
      pi[m].idx = 6; pi[m].seq = 1;
   end

   localparam int tests_per_sa = 50;
   localparam int num_tests = wid * tests_per_sa;
   localparam int cycle_limit = num_tests * wid * 2;

   uwire clk_reactive;
   reactivate ra(clk_reactive,clk);

   initial begin
      clk = 0;
      cycle = 0;

      fork
         forever #10 cycle += clk++;
         wait( done );
         wait( cycle >= cycle_limit )
           $write("*** Cycle limit exceeded, ending.\n");
      join_any;

      $finish();
   end

   initial begin

      // Number of test inputs (stimuli).
      //
      automatic int test_count = 0;

      done = 0;
      start = 1;

      @( posedge clk_reactive ); @( posedge clk_reactive );


      // Provide one test pattern per shift amount.
      //
      for ( int i=0; i<num_tests; i++ ) begin

         automatic int cyc_start = cycle;
         automatic int cyc_timeout = cycle + wid * 2;
```

```
      logic [wid-1:0] shadow_sout;
      int awaiting;
      automatic logic [wid_lg-1:0] amt_1 = i / tests_per_sa;

      amt = { amt_1[1:0], amt_1[wid_lg-1:2] };

      test_count++;

      for ( int p=0; p<wid; p+=32 ) sin[p+:32] = $random;


      shadow_sout = sin << amt;

      start = 1;
      @( posedge clk_reactive );
      start = 0;

      // Collect output as ready signals go to 1, or immediately
      // for non-sequential modules.
      //
      awaiting = pi.num();
      foreach ( pi[muti] ) begin
         automatic string mut = muti; // Bug workaround?
         fork begin
            while ( pi[mut].seq
                    && ready[pi[mut].idx] !== 1
                    && cycle < cyc_timeout )
              @( posedge clk_reactive );
            awaiting--;
            pi[mut].sout = sout[pi[mut].idx];
            pi[mut].cyc_tot += cycle - cyc_start;
         end join_none;
      end
      wait ( awaiting == 0 );

      // Check the output of each Module Under Test.
      //
      foreach ( pi[ mut ] )
        if ( shadow_sout !== pi[mut].sout ) begin
           pi[mut].err_count++;
           if ( pi[mut].err_count < 5 )
             $write
               ("%-20s wrong result for 0x%0h << %0d:  0x%0h != 0x%0h (correct)\n",
                mut, sin, amt, pi[mut].sout, shadow_sout);
        end

   end

   done = 1;

   foreach ( pi[ mut ] )
      $write("Ran %4d tests for %-15s, %4d errors found. Avg cyc %.1f\n",
             test_count, mut, pi[mut].err_count,
            pi[mut].seq ? real'(pi[mut].cyc_tot) / test_count : 1
```

```
                );
    end

endmodule

// cadence translate_on
```

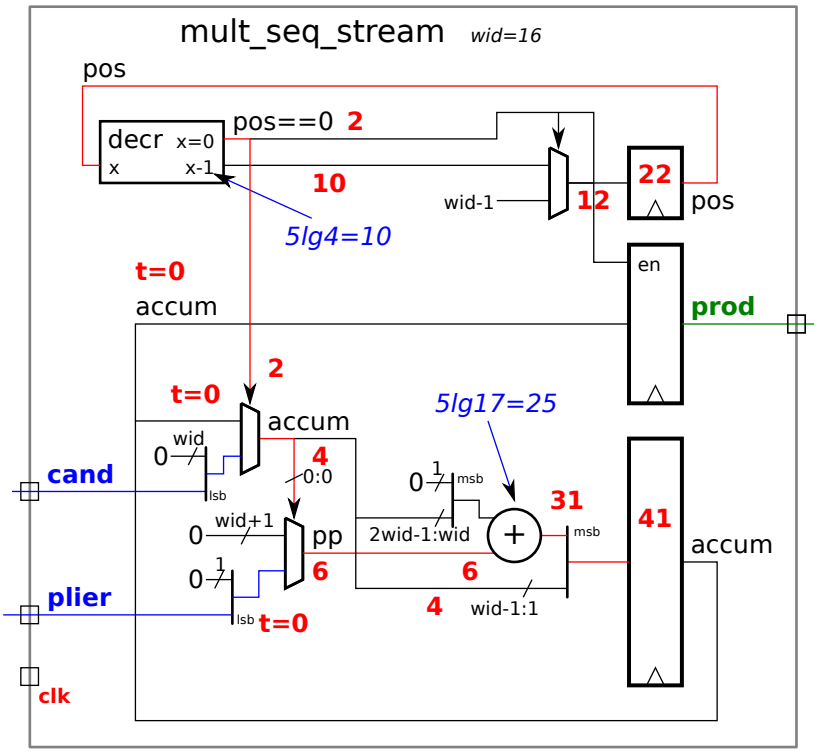
LSU EE 4755 Homework 5 Solution Due: 23 Oct 2015 17:00

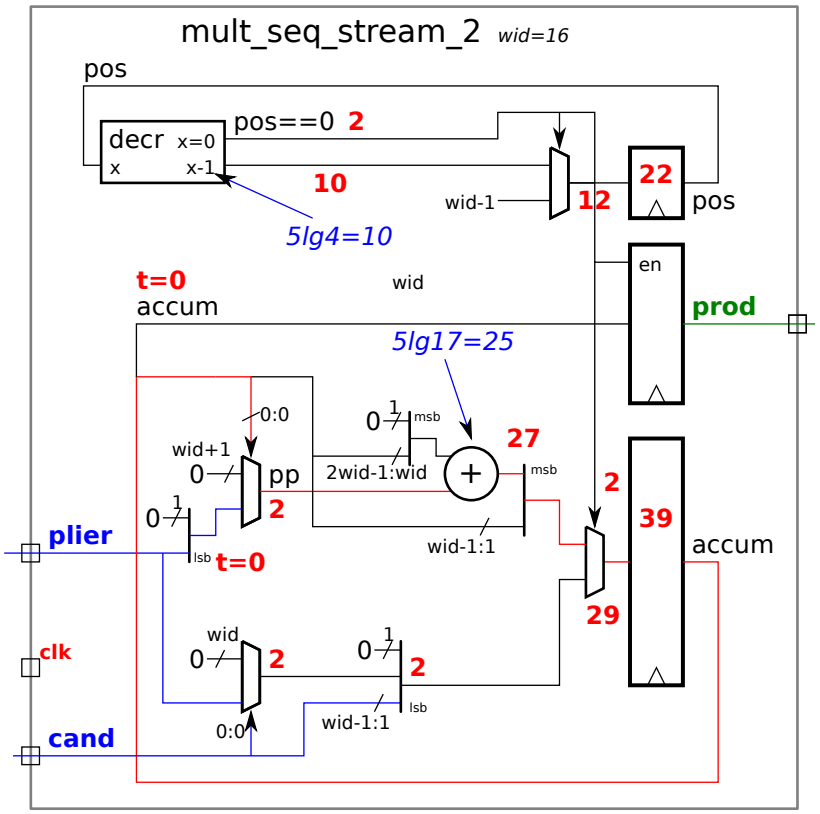
Problem 1: The homework Verilog file, `hw05.v`, contains something similar to the streamlined multiplier presented in class, `mult_seq_stream`, and even more streamlined versions of the multiplier, `mult_seq_stream_2`, and `mult_seq_stream_3`. These modules are reproduced at the end of this assignment. For an HTML version visit `http://www.ece.lsu.edu/koppel/v/2015/hw05.v.html`. *See the 2014 midterm exam for similar problems.*

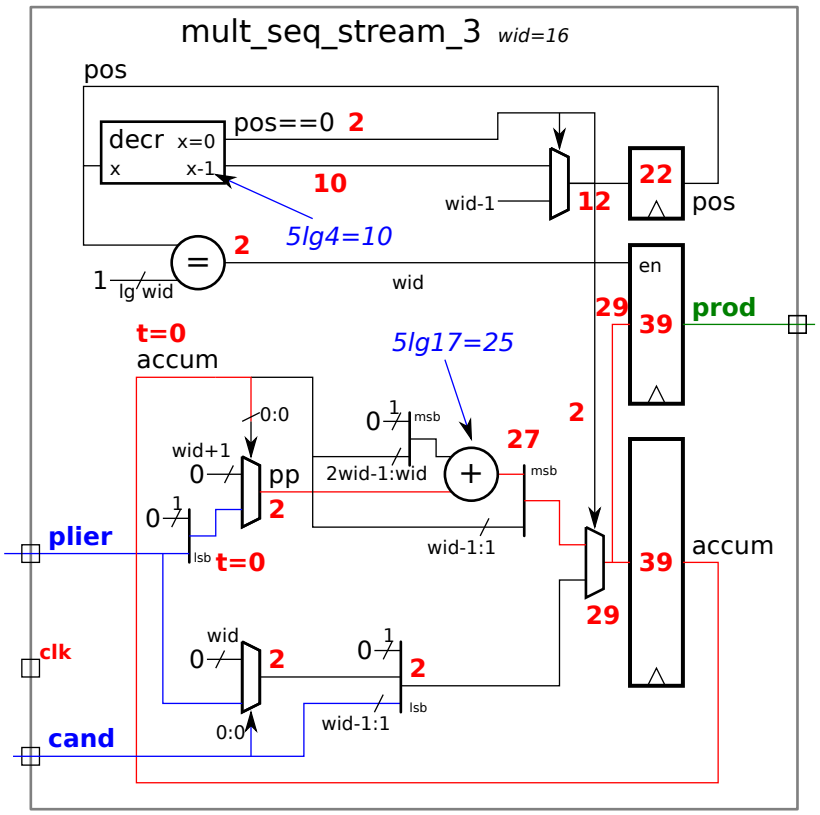
(*a*) Show the hardware that will be synthesized for each module for the default parameters. Show the module after optimization.

The synthesized hardware for each module appears below, and they also appear next to the respective Verilog descriptions at the end of this assignment. The red numbers show signal arrival times based on the assumptions given in the sub-problem below. The red wires show the critical path based on this analysis.

A `decr` unit has been used compute both `pos-1` and `pos==0`, under the assumption that it might be possible to share some hardware. An enable signal is used on the `prod` register.




mult_seq_stream_2 wid=16
pos
decr x=0
x x-1
pos==0 2
10
5lg4=10
wid-1
12
22
pos
t=0
accum
wid
en
prod
5lg17=25
0:0
wid+1
0
pp
2
0
1
lsb
t=0
plier
clk
cand
0
1
msb
2wid-1:wid
27
msb
wid-1:1
2
39
accum
29
wid
0
2
0:0
0
1
2
lsb
wid-1:1


mult_seq_stream_3 wid=16
pos
decr x=0
x x-1
pos==0 2
10
5lg4=10
wid-1
12
22
pos
=
2
1
lg wid
wid
en
29
39
prod
t=0
accum
5lg17=25
0:0
wid+1
0
pp
2
0
1
lsb
t=0
plier
clk
cand
0
1
msb
2wid-1:wid
27
msb
wid-1:1
2
39
accum
29
wid
0
2
0:0
0
1
2
lsb
wid-1:1

(*b*) Estimate the clock frequency of each module based on the following latencies:

Latch delay: 10 units. Multiplexor latency: 2 units. Latency of an n-bit adder: $5\lceil \lg n \rceil$ units. Latency of an n-input gate: $\lceil \lg n \rceil$ units. Let a unit be equal to 10 ps. *Note: The duration of a unit was not given in the original assignment.*

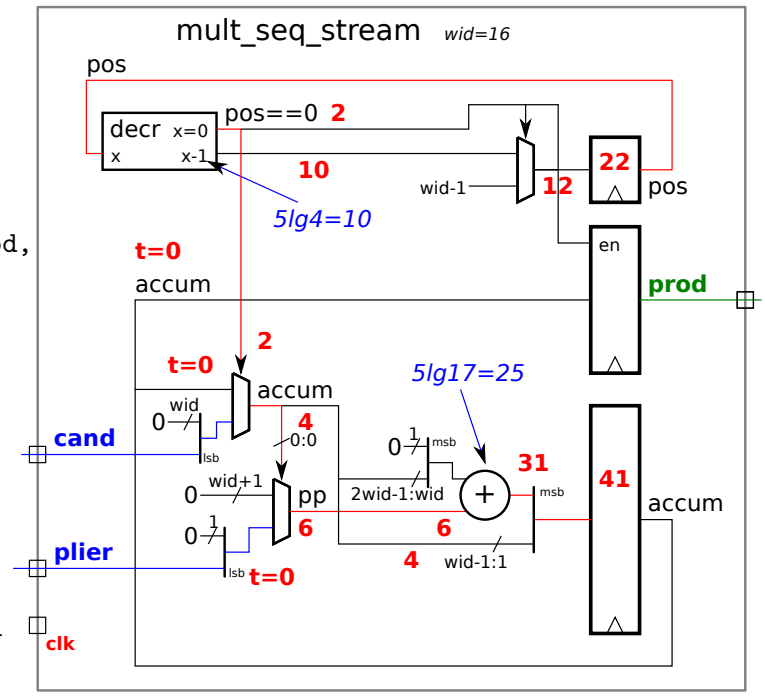
The timing analysis is shown in red on the three modules and the wires carrying the critical path are shown in red. This timing analysis strictly follows the guidelines above, using a $5\lceil \lg 17 \rceil = 25$ unit delay for the big adder. Realistically, that would be a 16-bit adder with a carry out. Solutions that used 20 rather than 25 units for the adder are correct.

For `mult_seq_stream` the critical path ends at `accum` with a period of 41 units or 410 ps. That would give a clock frequency of $\frac{1}{41}$ cycles per unit or 2.44 GHz.

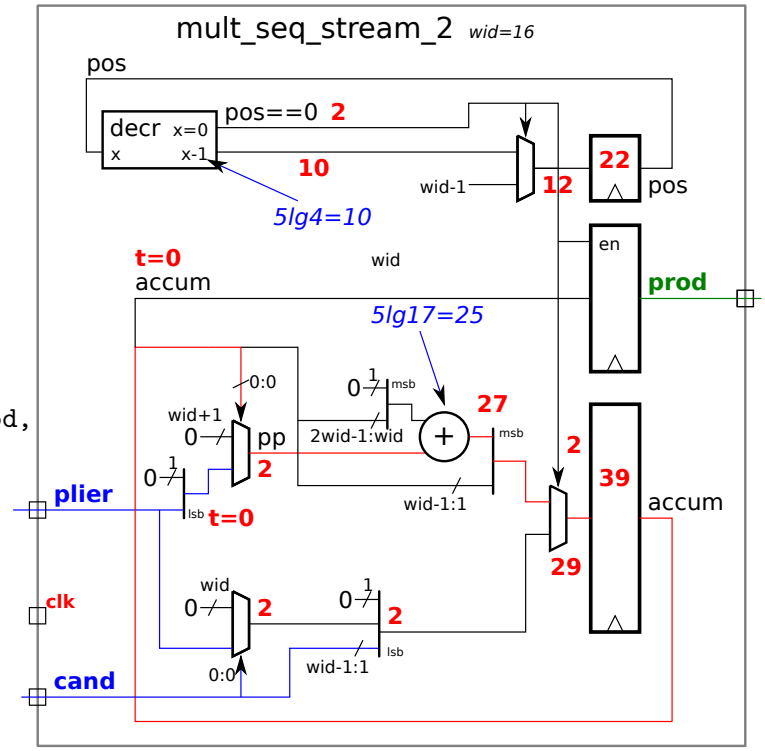
For `mult_seq_stream_2` and `mult_seq_stream_3` the critical path is 2 units shorter, at 39 units. This is because the big adder uses the `accum` signal right out of the register outputs, in contrast to `mult_seq_stream` in which the particular `accum` to use must be routed through a mux based on a `pos==0` select, adding delay. The clock frequency for these two modules would be 2.56 GHz.

(*c*) Why would module `mult_seq_stream_3` provide a result in less time than the other two, even assuming that the clock frequency for all the modules was the same?

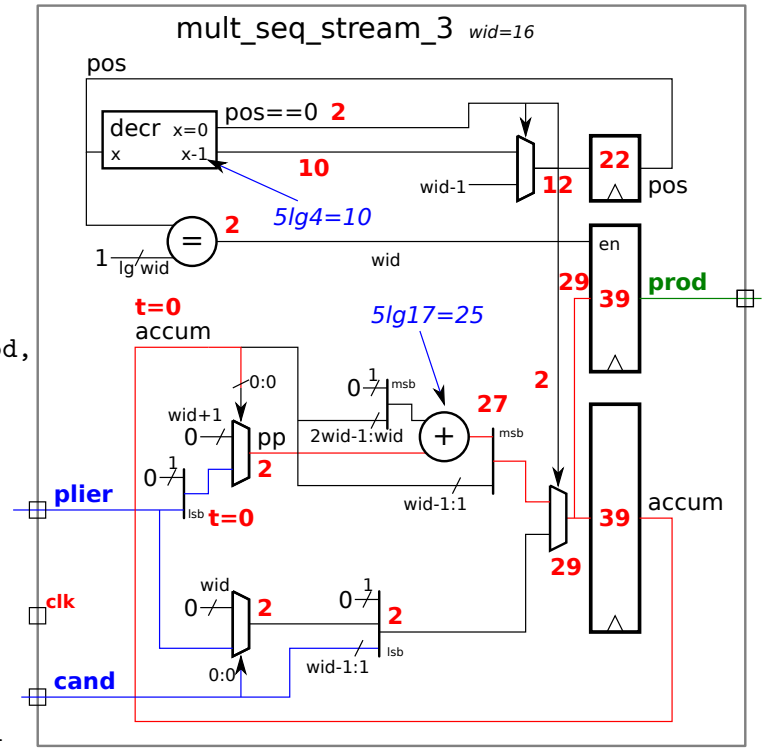
The product is available one cycle earlier because it is written to `prod` from the output of the big adder rather than from `accum`.




```
module mult_seq_stream
 #( int wid = 16 )
  ( output logic [2*wid-1:0] prod,
    input logic [wid-1:0] plier,
    input logic [wid-1:0] cand,
    input clk);

   localparam int wlog =
     $clog2(wid);

   logic [wlog-1:0] pos;
   logic [2*wid-1:0] accum;

   always @( posedge clk ) begin

      logic [wid:0] pp;

      if ( pos == 0 ) begin

         prod = accum;
         accum = cand;
         pos = wid - 1;

      end else begin

         pos--;

      end

      // Note: the multiplicand is in the lower bits of the accumulator.
      //
      pp = accum[0] ? { 1'b0, plier } : 0;

      // Add on the partial product and shift the accumulator.
      //
      accum = { { 1'b0, accum[2*wid-1:wid] } + pp, accum[wid-1:1] };

   end

endmodule
```




```
module mult_seq_stream_2
 #( int wid = 16 )
  ( output logic [2*wid-1:0] prod,
    input logic [wid-1:0] plier,
    input logic [wid-1:0] cand,
    input clk);

   localparam int wlog =
     $clog2(wid);
   logic [wlog-1:0] pos;
   logic [2*wid-1:0] accum;

   always @( posedge clk ) begin

      if ( pos == 0 ) begin

         prod = accum;
         accum = { 1'b0, cand[0] ? plier : wid'(0), cand[wid-1:1] };
         pos = wid - 1;

      end else begin

         logic [wid:0] pp;

         // Note: the multiplicand is in the lower bits of the accumulator.
         //
         pp = accum[0] ? plier : 0;

         // Add on the partial product and shift the accumulator.
         //
         accum = { { 1'b0, accum[2*wid-1:wid] } + pp, accum[wid-1:1] };

         pos--;

      end

   end

endmodule
```




```
module mult_seq_stream_3
 #( int wid = 16 )
  ( output logic [2*wid-1:0] prod,
    input logic [wid-1:0] plier,
    input logic [wid-1:0] cand,
    input clk);

   localparam int wlog =
     $clog2(wid);

   logic [wlog-1:0] pos;
   logic [2*wid-1:0] accum;

   always @( posedge clk ) begin

      if ( pos == 0 ) begin

         accum = { 1'b0, cand[0] ? plier : wid'(0), cand[wid-1:1] };
         pos = wid - 1;

      end else begin

         logic [wid:0] pp;

         // Note: the multiplicand is in the lower bits of the accumulator.
         //
         pp = accum[0] ? plier : 0;

         // Add on the partial product and shift the accumulator.
         //
         accum = { { 1'b0, accum[2*wid-1:wid] } + pp, accum[wid-1:1] };

         if ( pos == 1 ) prod = accum;

         pos--;

      end

   end

endmodule
```

LSU EE 4755 Homework 6 Solution Due: 2 December 2015

The solution code has been placed in /home/faculty/koppel/pub/ee4755/hw/2015f/hw06/hw06.v and an htmlized version is at `http://www.ece.lsu.edu/koppel/v/2015/hw06sol.v.html`, the original code in htmlized form can be found at `http://www.ece.lsu.edu/koppel/v/2015/hw06.v.html`.

Problem 0: The homework Verilog file, `hw06.v`, contains something similar to the integer compression modules presented in class. (Follow the homework workflow instructions on the course procedures page to get a copy of the assignment package.) These modules compress an ASCII character stream by substituting a binary-encoded integer for a string of ASCII digits. These modules were based on 2014 Homework 4. Feel free to look at that assignment an solution for help.

Module `icomp_none` is a version of the module that does no compression at all. It does though implement the handshaking protocol so that characters can be passed from input to output. This module can be studied to help understand how the others work.

Module `icomp_2cyc` is one of the compression modules covered in class. It computes the encoded value in stage 0, and checks for overflow in stage 1. Don't modify this module, save if for reference. Module `icomp_sol` is initially identical to `icomp_2cyc`, but it should be modified as part of this assignment.

The testbench is set to simulate `icomp_sol` on a sample test string. At the end it will report the amount of compression and whether there was any errors. The testbench also prints out a trace showing some module inputs and outputs and the status of internal signals. Examine the testbench code to see how this is done and feel free to modify it to add signals of your own. A more detailed trace of execution can be obtained using the SimVision gui. To start that use the command `irun hw06.v -gui`. See `http://www.ece.lsu.edu/koppel/v/v/s/SimVisionIntro.pdf` for documentation. (On campus access only without password.)

The synthesis script will synthesize the modules `icomp_2cyc` and `icomp_sol`. Use the synthesis script to make sure that your designs are synthesizable and to determine their cost and performance.

(There is nothing to turn in for this assignment.)

Problem 1: In module `icomp_sol` there is a declaration of a variable named `val_encode_size_1`, but no uses of that variable. Add code to that module so that `val_encode_size_1` is set to the number of bytes that are needed for the number currently in the register `val_encode_1`. For example, if `val_encode_1` has a 0, then `val_encode_size_1` should be 0. If `val_encode_1` has a 123 then `val_encode_size_1` should be 1 (one byte), if `val_encode_1` has a 300 then `val_encode_size_1` should be 2 (for 2 bytes), etc.

To help with your solution add code to the testbench to show the value of this variable.

The solution appears below. The idea is to check each byte of `val_encode_1`, from least significant to most significant. If the byte is non-zero tentatively set `val_encode_size_1` to the byte position (starting at one for the least-significant byte). Note that `val_encode_1` is declared as a two-dimensional packed array, and so the expression `val_encode_1[i]` evaluates to the value of byte number `i` (with 0 being least significant, see the declaration).

```
logic [max_chars:0][7:0] val_encode_1;
logic [mc_bits:0] val_encode_size_1;
always_comb begin
   val_encode_size_1 = 0;
   for ( int i=0; i<max_chars; i++ )
     if ( val_encode_1[i] ) val_encode_size_1 = i + 1;
end
```

Problem 2: Modify module `icomp_sol` so that a group of ASCII digits is compressed into the smallest number of bytes needed, up to `max_chars`. For example, if `max_chars` is 4 then just use one byte to compress 200, two bytes for 4000, and for 1234567890123 use a four-byte integer (for 1234567890) followed by a one byte integer (for 123).

Precede the compressed integer by the character 128 plus the number of bytes in the compressed number. For example, if the compressed value takes two bytes then where the first character of the uncompressed value would go emit a 130, then the next two characters should be the compressed number. (See how `char_out` is assigned in the unmodified code.)

To solve this problem you'll need to understand how the existing code works, how to interpret the trace output provided by the simulator, and how to use the SimVision waveform viewer. Random guesses based on a vague understanding will get you nowhere.

- The module should be written for arbitrary values of `max_chars`.
- Make sure that the testbench is not reporting errors.
- Make sure that your module is compressing the string.

In the original module integers were encoded into `max_chars` bytes. So that the module can now encode integers into sizes from 1 up to `max_chars` bytes the following must be changed:

Encoding Acceptance: The hardware that decides whether to accept an encoded integer must now compare the ASCII length (`ascii_int_len`) to the actual encoded size (`val_encode_size_1`), not to `max_chars`. See `Changed Line` below.

```
wire use_encoding = end_encoding
     && ( ascii_int_len > 2 )                  /// Changed Line
     && ( !val_wait_full || end_draining );
```

Tail Changes: The position for writing incoming characters into storage is `tail`. Ordinarily `tail` is incremented each time a character is read. But because an encoded integer takes less space than the ASCII version `tail` must be adjusted after the last character of an encoded integer is encountered. In the original code the adjusted tail value is found by adding the starting point of the ASCII string (`tail_at_enc_start_1`) to `max_chars` plus a possible overflow adjustment. In the solution `max_chars` is replaced by `val_encode_size_1`. (The overflow adjustment adds an extra one to the tail because the tail is being updated one cycle late.)

```
wire [size_lg:1] tail_adj =
    tail_at_enc_start_1 + val_encode_size_1 + overflow_1;
```

Head/Char Out Changes: Module output `char_out` can connect to either `storage` (where the ASCII characters are stored), the escape character (a constant value in the original module), or `val_wait` (the encoded integer). In the original code control logic would connect `char_out` to `val_wait` until `max_chars` characters were read. In the modified module it connects `char_out` to `val_wait` until `val_encode_size_1` characters were read.

In the original code each element of array `esc_here` was one bit, indicating that the corresponding ASCII character in `storage` was the start of an ASCII string that should be replaced by an encoded integer. In the solution each element of `esc_here` indicates how many bytes are in the encoded integer (a 0 means that an encoded integer does not start here). The solution excerpt below shows the new declaration for `esc_here` and how `esc_here` gets written:

```
/// SOLUTION HIGHLIGHTS -- SURROUNDING CODE REMOVED

/// SOLUTION -- Problem 2
//  Increase the size of the "escape here" marker from 1 bit to
//  mc_bits. Its value now indicates the size of the encoded
```

```
//  integer in bytes.
logic [mc_bits-1:0] esc_here [size];

   /// SOLUTION -- Problem 2
   //
   //  Write the size of the encoded integer into the esc_here array.
   //  (Previously we just wrote a 1, to indicate that an encoded
   //  integer starts at this position.)
   //
always_ff @( posedge clk )
   if ( use_encoding ) esc_here[tail_at_enc_start_1] <= val_encode_size_1;
```

Head/Char Out Changes continued: The variable **drain_idx** indicates the byte position in **val_wait** that should be sent to **char_out**. In the original code it was initialized to **max_chars-1**, an elaboration-time constant. In the solution it is set to **esc_here[head]-1**, see the first excerpt below. The final change is to change the escape character. In earlier classroom examples the escape character was a constant, **Char_escape**. In this assignment the escape character should be set to the sum of **Char_escape** and the size of the encoded integer. In the original code, that's still a constant because both **Char_escape** and **max_chars** are elaboration-time constants. But in the solution the encoded size can vary, so we must add the actual encoded size, **esc_here[head]** to **Char_escape**, that appears in the second excerpt below.

```
   /// SOLUTION -- Problem 2
   //
   //  Initialize drain_idx with one minus the size of
   //  the encoded integer, rather than max_chars - 1.
   //
   drain_idx <= start_draining  ?  esc_here[head] - 1 :
                drain_idx > 0   ?  drain_idx - 1 :
                                   0;
```

```
/// SOLUTION -- Problem 2
//
//  When we reach an encoded integer output the escape character
//  plus the size of the encoded integer.
//
assign char_out =
       start_draining  ?  Char_escape + esc_here[head] :
       draining        ?  val_wait[drain_idx] :
                          storage[head];
```

24 Fall 2014 Solutions

```
////////////////////////////////////////////////////////////////////////////////
//
 /// LSU EE 4755 Fall 2014 Homework 1 -- SOLUTION
//

 /// Assignment http://www.ece.lsu.edu/koppel/v/2014/hw01.pdf


`default_nettype none


////////////////////////////////////////////////////////////////////////////////
///  Problem 1
//
 /// Logical Right Shift Module 1
//
//   The module below performs a logical right shift of a 16-bit
//   quantity.
//
//   [x]  Fix the module, the testbench should not report errors.
//   [x]  Don't substantially change the way the code works.
//   [x]  Don't try to make the code synthesizable.
//   [x]  Don't use shift (<<) or concatenation operators ({}) ..
//        .. assign shifted bit-by-bit as the code already does.


// cadence translate_off

module shift_right1
   ( output logic [15:0] shifted,
     input wire [15:0] unshifted,
     input wire [3:0] amt );

   /// Problem 1 solution goes in this module.

   localparam int width = 16;

   always @* begin

      automatic int limit = width - amt;

      for ( int i=0; i<limit; i++ ) shifted[i] = unshifted[i+amt];

      /// SOLUTION
      //  Just zero the "vacated" bits.
      //
      for ( int i=limit; i<width; i++ ) shifted[i] = 0;

   end

endmodule

// cadence translate_on


////////////////////////////////////////////////////////////////////////////////
///  Problem 2
//
 /// Logical Right Shift Module 2
//
//   The module below performs a logical right shift of a 16-bit
//   quantity.
//
```

```
//   [x]  Complete module shift_right2, the testbench should not report errors.


// Perform Two Possible Shifts:  by 0 bits or by fsamt bits.
//
module shift_right_fixed
   ( output wire [15:0] shifted,
     input wire [15:0] unshifted,
     input wire shift );

   // Problem 2: DON'T modify this module.

   // (Fixed) Shift Amount
   //
   parameter int fsamt = 3;

   // If shift is true shift by fsamt, otherwise don't shift.
   //
   assign    shifted =  shift  ?  unshifted >> fsamt  :  unshifted;

endmodule

module shift_right2
  ( output wire [15:0] shifted,
    input wire [15:0] unshifted,
    input wire [3:0] amt );

   /// Problem 2 solution goes in this module.

   /// SOLUTION
   //
   //  Declare wires to interconnect the modules.
   //
   uwire [15:0]  s8, s4, s2;

   shift_right_fixed #(8) sm8
     ( .shifted(s8), .unshifted(unshifted), .shift(amt[3]) );

   /// SOLUTION
   //
   //  Instantiate three more modules and connect them.
   //  Note: You don't have to use named ports.

   shift_right_fixed #(4) sm4 (s4, s8, amt[2]);
   shift_right_fixed #(2) sm2 (s2, s4, amt[1]);
   shift_right_fixed #(1) sm1 (shifted, s2, amt[0]);

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Testbench Code
//
//  The code below instantiates shift_right1 and shift_right2,
//  provides test inputs and verifies the outputs.
//
//  The testbench may be modified to facilitate your solution. Of
//  course, the removal of tests which your module fails is not
//  considered a correct solution. (The idea is to put in tests which
//  make it easier to determine what the problem is, for example, test
//  inputs that are all 0's or all 1's.)


// cadence translate_off
```

```
module testbench();

   uwire logic [15:0] sout1, sout2;
   logic [15:0] sin;
   logic [3:0] amt;

   shift_right1 my_sr1(sout1, sin, amt);
   shift_right2 my_sr2(sout2, sin, amt);

   // Width of shifters' input and output.
   // The parameter is used only by this testbench.
   //
   localparam int width = 16;
   //
   // To keep things simple the shifter modules themselves are written
   // with a hardcoded width of 16 bits. That's bad style since
   // changing the width would be tedious and error prone. The
   // hardcoded widths are used in this first homework assignment only
   // to keep things simple. (The shifter modules could have used a
   // parameter to specify the width or a user-defined type.)

   // Provide names for the modules for use in error messages.
   //
   localparam string name[2] = '{"Prob 1", "Prob 2"};

   initial begin

      // Count of errors for each module.
      //
      automatic int err_count[2] = '{0,0};
      //
      // Note: The automatic qualifier is needed so that the initialization
      // could appear on the same line as the declaration.

      // Number of test inputs (stimuli).
      //
      automatic int test_count = 0;

      // Provide one test pattern per shift amount.
      //
      for ( int i=0; i<width; i++ ) begin

         int shadow_sout;
         test_count++;

         sin = $random;
         amt = i;

         shadow_sout = sin >> amt;

         #1;

         // Check the output of each Module Under Test.
         //
         foreach ( name[ mut ] ) begin

            automatic logic [15:0] sout = mut == 0 ? sout1 : sout2;

            if ( shadow_sout !== sout ) begin
               err_count[mut]++;
               if ( err_count[mut] < 5 )
                 $display
                   ("MUT %s wrong result for %h >> %d:  %h != %h (correct)\n",
                    name[mut], sin, amt, sout, shadow_sout);
```

```
            end

          end

        end

        $display("Ran %d tests, %d, %d errors found.\n",
                 test_count, err_count[0], err_count[1]);

    end

endmodule

// cadence translate_on
```

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2014 Homework 2
//

 /// Assignment http://www.ece.lsu.edu/koppel/v/2014f/hw02.pdf

 /// Instructions:
  //
  // (1) Find the undergraduate workstation laboratory, room 126 EE
  //     Building.
  //
  // (2) Locate your account.  If you did not get an account please
  //     E-mail: koppel@ece.lsu.edu
  //
  // (3) Log in to a Linux workstation.
  //     The account should start up with a WIMP interface (windows, icons,
  //     mouse, pull-down menus)  ( :-) ) but one or two things need
  //     to be done from a command-line shell.  If you need to brush up
  //     on Unix commands follow http://www.ece.lsu.edu/koppel/v/4ltrwrd/.
  //
  // (4) If you haven't already, follow the account setup instructions here:
  //     http://www.ece.lsu.edu/koppel/v/proc.html
  //
  // (5) Copy this assignment, local path name
  //     /home/faculty/koppel/pub/ee4755/hw/2014f/hw02
  //     to a directory ~/hw02 in your class account. (~ is your home
  //     directory.) Use this file for your solution.
  //
  // (6) Find the problems in this file and solve them.
  //
  //     Your entire solution should be in this file.
  //
  //     Do not change module names.
  //
  // (7) Your solution will automatically be copied from your account by
  //     the TA-bot.


 /// Additional Resources
  //
  // Verilog Documentation
  //    The Verilog Standard
  //      http://standards.ieee.org/getieee/1800/download/1800-2012.pdf
  //    Introductory Treatment (Warning: Does not include SystemVerilog)
  //      Brown & Vranesic, Fundamentals of Digital Logic with Verilog, 3rd Ed.
  //
  // Account Setup and Emacs (Text Editor) Instructions
  //      http://www.ece.lsu.edu/koppel/v/proc.html
  //      To learn Emacs look for Emacs tutorial.
  //
  // Unix Help
  //      http://www.ece.lsu.edu/koppel/v/4ltrwrd/


////////////////////////////////////////////////////////////////////////////////
/// Behavioral Multipliers


module mult_behav_1
  #(int wid = 16)
   (output logic[2*wid-1:0] prod, input logic[wid-1:0] plier, cand);
   assign prod = plier * cand;
```

```
endmodule


module mult_behav_2
  #(int wid = 16)
   (output logic[2*wid-1:0] prod, input logic[wid-1:0] plier, cand);

   always @* begin

      prod = 0;

      for ( int i=0; i<wid; i++ ) if ( plier[i] ) prod = prod + ( cand << i );

   end

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Problem 2: Linear Multiplier

 /// Simple Adder, Don't Modify
module good_adder#(int w=16)(output [w:1] s, input [w:1] a,b);
   assign s = a + b;
endmodule


module mult_linear
  #(int wid = 16)
   (output logic[2*wid-1:0] prod, input logic[wid-1:0] plier, cand);

   /// Problem 2 Solution Goes Here.
   //  This module should be a structural version of mult_behav_2,
   //   using generate statements to instantiate good_adder.


   /// SOLUTION BELOW

   logic [2*wid-1:0] rsum [wid-1:-1];
   logic [2*wid-1:0] pp   [wid-1:0];

   assign rsum[-1] = 0;

   for ( genvar i=0; i<wid; i++ ) begin
      assign pp[i] = plier[i] ? cand << i : 0;
      good_adder #(2*wid) adder(rsum[i], rsum[i-1], pp[i] );
   end

   assign    prod = rsum[wid-1];

endmodule

////////////////////////////////////////////////////////////////////////////////
/// Problem 3: Tree Multiplier


 /// Problem 3 Solutions
 //
 //  Several solutions appear below to Problem 3. The easy to
 //  understand solution is mult_tree_simple. Module mult_tree is
 //  compact (does not require alot of Verilog code). Module
 //  mult_tree_rec shows a recursive implementation. Module
 //  mult_tree_better uses a cost saving technique.
```

```
//
/// mult_tree
//  This one is the shortest.  The tree is constructed using
//  a single loop.
//
/// mult_tree_rec
//  A recursive version.  The cost and performance will
//  not be very good unless synthesized with option "-effort high"
//  because without that option the synthesis program synthesizes
//  modules without taking into account how they are instantiated.
//  Among other things, that means the synthesis program can't eliminate
//  unused wires.
//
/// mult_tree_simple
//  Maybe the easiest to understand. The tree is constructed using
//  two nested loops, the outer loop iterates over tree levels
//  and the inner loop iterates over adders within a level.
//
/// mult_tree_better
//  This is like mult_tree_simple, but instead of shifting the
//  multiplicand, the intermediate sums are shifted.


module mult_tree
  #(int wid = 16)
   (output logic[2*wid-1:0] prod, input logic[wid-1:0] plier, cand);

   localparam int widp2 = 1 << $clog2(wid);

   /// SOLUTION BELOW to Problem 3
   //
   //  This is one of several solutions to Problem 3.

   logic [2*wid-1:0] rsum [2*wid-1:0];
   localparam int mask = 2*wid-1;

   // Compute partial products.
   //
   for ( genvar i=0; i<wid; i++ )
     assign rsum[i] = plier[i] ? cand << i : 0;

   // Add partial products together.
   //
   for ( genvar i=wid; i<2*wid-1; i++ )
     good_adder #( 2*wid ) adder
                ( rsum[i],
                  rsum[ mask &   (i<<1)       ],   // Left child.
                  rsum[ mask & ( (i<<1) + 1 ) ]    // Right child.
                  );

   assign    prod = rsum[2*wid-2];

endmodule

module mult_tree_rec
  #( int wid_plier = 16,
     int wid_cand = wid_plier )
   ( output logic [2*wid_plier-1:0] prod,
     input logic [wid_plier-1:0] plier,
     input logic [wid_cand-1:0] cand);

   localparam int wid_cr_h = wid_cand / 2;
   localparam int wid_cr_l = wid_cand - wid_cr_h;

   generate
```

```
      if ( wid_cand == 1 ) begin

         assign prod = cand[0] ? plier : 0;

      end else begin

         wire    logic [2*wid_plier-1:0] prod_h, prod_l;

         mult_tree_rec #(wid_plier, wid_cr_h) m_h
           (prod_h, plier, cand[wid_cand-1:wid_cr_l] );

         mult_tree_rec #(wid_plier, wid_cr_l) m_l
           (prod_l, plier, cand[wid_cr_l-1:0] );

         good_adder #(2*wid_plier) adder
           (prod, prod_h << wid_cr_l, prod_l);

      end

   endgenerate

endmodule



module mult_tree_simple
  #(int wid = 16)
   (output logic[2*wid-1:0] prod, input logic[wid-1:0] plier, cand);

   localparam int levels = $clog2(wid);

   logic [2*wid-1:0] rsum [2*wid-1:0][levels:0];

   for ( genvar lev=0; lev<levels; lev++ ) begin

      localparam int siblings = 1 << lev;

      for ( genvar i=0; i<siblings; i++ )

        good_adder #(2*wid) adder
                    ( rsum[i][lev],
                      rsum[i*2][lev+1],
                      rsum[i*2+1][lev+1] );

   end

   for ( genvar i=0; i<wid; i++ )
     assign rsum[i][levels] = plier[i] ? cand << i : 0;

   assign   prod = rsum[0][0];

endmodule


module mult_tree_better
  #(int wid = 16)
   (output logic[2*wid-1:0] prod, input logic[wid-1:0] plier, cand);

   localparam int levels = $clog2(wid);

   logic [2*wid-1:0] rsum [2*wid-1:0][levels:0];

   for ( genvar lev=0; lev<levels; lev++ ) begin
```

```
      localparam int siblings = 1 << lev;
      localparam int shift = 1 << levels - lev - 1;

      for ( genvar i=0; i<siblings; i++ )

        good_adder #(2*wid) adder
                   ( rsum[i][lev],
                     rsum[i*2+1][lev+1] << shift,
                     rsum[i*2][lev+1]);

   end

   // Notice that the multiplicand is not shifted here.
   //
   for ( genvar i=0; i<wid; i++ )
     assign rsum[i][levels] = plier[i] ? cand : 0;

   assign   prod = rsum[0][0];

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Testbench Code

// cadence translate_off

module testbench;

   localparam int wid = 64;
   localparam int num_tests = 1000;
   localparam int NUM_MULT = 7;
   localparam int err_limit = 4;

   logic [wid-1:0] plier, cand;
   logic [2*wid-1:0] prod[NUM_MULT];

   mult_behav_1 #(wid) mb1(prod[0], plier, cand);
   mult_behav_2 #(wid) mb2(prod[1], plier, cand);
   mult_linear  #(wid) ms1(prod[2], plier, cand);
   mult_tree    #(wid) ms2(prod[3], plier, cand);
   mult_tree_rec #(wid) ms3(prod[4], plier, cand);
   mult_tree_simple #(wid) ms4(prod[5], plier, cand);
   mult_tree_better #(wid) ms5(prod[6], plier, cand);

   string names[] = '{"Behav_1","Behav_2","Linear", "Tree",
                      "Tree Rec", "Tree Simple", "Tree Average"};
   int err_cnt[NUM_MULT];
   int tests[$] = {1,1, 1,2,  1,32,  32, 1};

   initial begin

      for ( int i=0; i<num_tests; i++ ) begin

         plier = tests.size() ? tests.pop_front() : $random();
         cand = tests.size() ? tests.pop_front() : $random();

         #1;

         for ( int mut=1; mut<NUM_MULT; mut++ ) begin

            if ( prod[0] !== prod[mut] ) begin

               err_cnt[mut]++;
```

```
            if ( err_cnt[mut] < err_limit )
              $display("Error in %s test %4d:  %d != %d (correct)\n",
                       names[mut], i, prod[mut], prod[0]);
          end

        end

      end

      for ( int mut=1; mut<NUM_MULT; mut++ ) begin

        $display("Mut %s, %d errors (%.1f%% of tests)\n",
                 names[mut], err_cnt[mut],
                 100.0 * err_cnt[mut]/real'(num_tests) );

      end


  end

endmodule

// cadence translate_on
```

LSU EE 4755 **Homework 3** Solution **Due: 24 October 2014**

Updated 7 November 2014, 13:49:47 CST

The Homework 3 code package contains a simple behavioral multiplier and several sequential multipliers. It also contains a synthesis script in file `syn.cmd`.

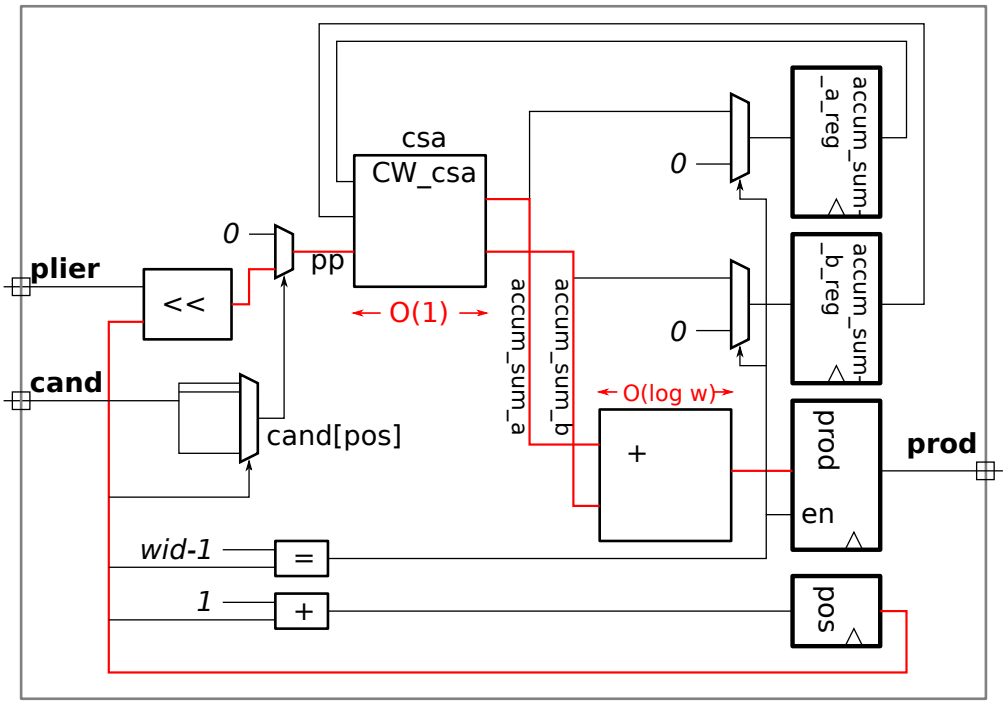
Problem 0: Copy the code package from `/home/faculty/koppel/pub/ee4755/hw/2014f/hw03`. Verify that everything is working by running the simulation on the unmodified file. It should report a 0% error rate for all modules.

Problem 1: The module `mult_seq_csa` is a sequential multiplier that instantiates an adder, however unlike `mult_seq_ga` shown in class, `mult_seq_csa` instantiates a carry-save adder from the Chipware library, `CW_csa`. The carry save adder computes the sum of three integers, `a`, `b`, and `c` (those are the port names). It produces two sums, which we'll call `sum_a` and `sum_b` (the port names for these are `carry` and `sum`). All of these ports are w bits wide, where w is a parameter. The actual sum of `a`, `b`, and `c` is obtained by adding together outputs `sum_a` and `sum_b` using a conventional adder. Carry save adders are used when there many integers to be added. Some arrangement (linear, tree) of many carry-save adders will produce a `sum_a` and `sum_b`, which will be added by a single conventional (called carry-propagate) adder.

The advantage of a carry save adder is that it can compute a sum of w-bit numbers in $O(1)$ time (the amount of time is not affected by w), which of course is much better than the $O(w)$ time for a ripple adder or the $O(\log w)$ time for much more expensive carry look-ahead adders. The performance advantage of a CSA is lost for `mult_seq_csa` because the module only computes one partial product at a time.

(a) Sketch the hardware that will be synthesized for `mult_seq_csa`. Show the carry-save adder and other major units as boxes, but be sure to show registers, multiplexors, and other such components. **Do not** show the actual output produced by an actual synthesis program. (It's okay if you look at a synthesis program's output.)

The hardware appears below. In the diagram the critical path is shown in red. Notice that the critical path goes through *both* the CSA and conventional adders.



(b) Based on this sketch of synthesized hardware, explain why the benefit of using a CSA is lost. Also explain how the module can be made a little faster (with a small change), but is still not a good way to use a CSA.

The clock frequency is based on the (longest) critical path. For the module the critical path is the *sum* of the delay through the CSA and carry-propagate (regular) adder. If a carry-propagate adder were used

Problem 2: Module `mult_seq_csa_m` initially contains the m-partial-products-per-cycle module that we did in class. In this problem modify it to use CSA's, and avoid the issue identified in the previous problem.

(a) Modify `mult_seq_csa_m` so that it uses the carry-save adder to compute m partial products per cycle. Use `generate` statements to instantiate the CSA's, and of course, connect them appropriately. (In class we used generate statements for the pipelined adder to instantiate stages, that code is in `mult_pipe_ia` in the same file as the assignment.)

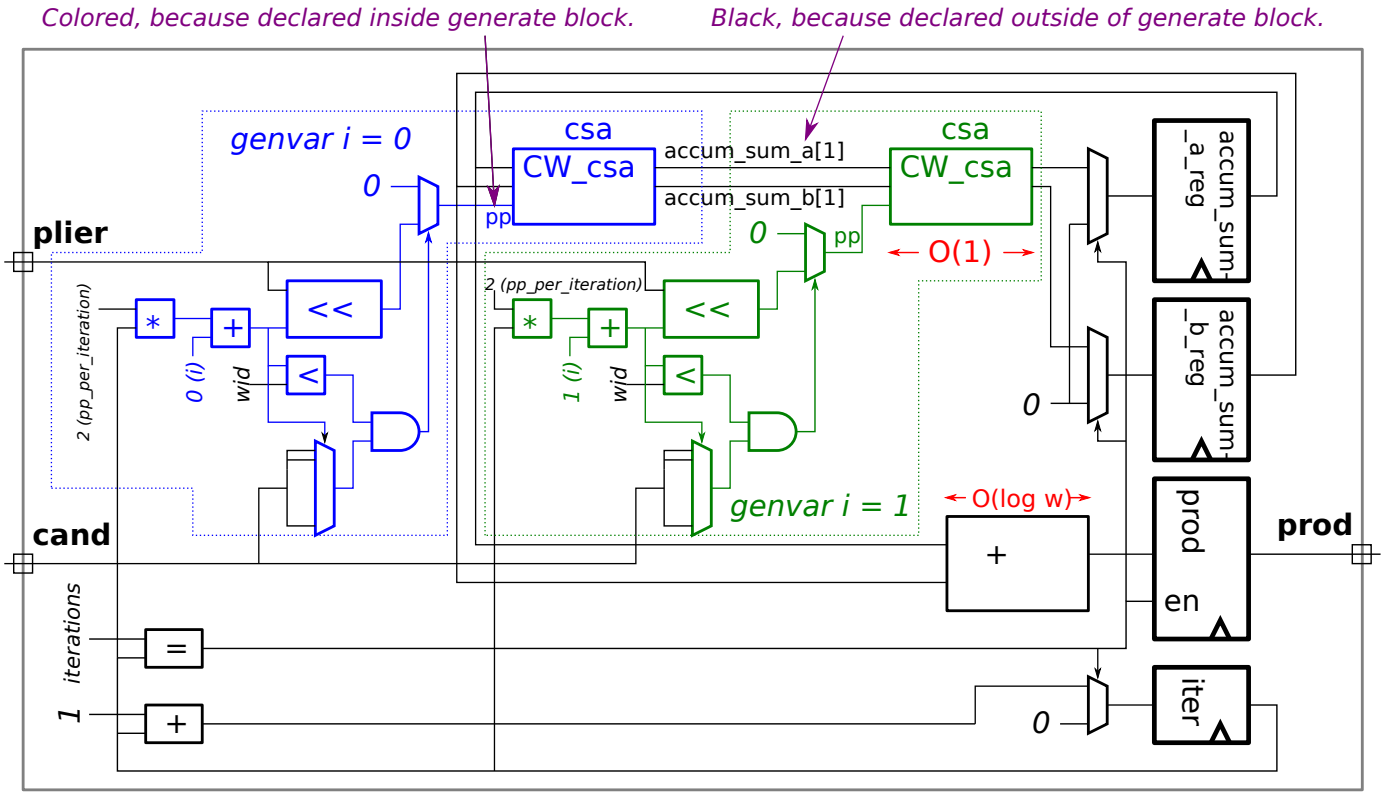
Solution appears below.

```
module mult_seq_csa_m #( int wid = 16, int pp_per_cycle = 2 )
   ( output logic [2*wid-1:0] prod,
     input logic [wid-1:0] plier,   input logic [wid-1:0] cand,   input clk);

   localparam int iterations = ( wid + pp_per_cycle - 1 ) / pp_per_cycle;
   localparam int iter_lg = $clog2(iterations);
   localparam int wid_lg = $clog2(wid);

   logic [iter_lg:0] iter;
   wire [2*wid-1:0] accum_sum_a[0:pp_per_cycle], accum_sum_b[0:pp_per_cycle];
   logic [2*wid-1:0] accum_sum_a_reg, accum_sum_b_reg;

   initial iter = 0;
   assign           accum_sum_a[0] = accum_sum_a_reg;
   assign           accum_sum_b[0] = accum_sum_b_reg;

   for ( genvar i=0; i<pp_per_cycle; i++ ) begin

      wire [wid_lg:1] pos = iter * pp_per_cycle + i;
      wire            co; // Unconnected.

      // The "pos < wid" below is needed in case wid is not an integer multiple of pp_per_cycle.
      wire [2*wid-1:0] pp = pos < wid && cand[pos] ? plier << pos : 0;

      CW_csa #(2*wid) csa
        ( .sum(accum_sum_a[i+1]), .carry(accum_sum_b[i+1]), .co(co),
          .a(accum_sum_a[i]), .b(accum_sum_b[i]), .c(pp), .ci(1'b0) );
   end

   always @( posedge clk ) if ( iter == iterations ) begin

         prod <= accum_sum_a_reg + accum_sum_b_reg;
         accum_sum_a_reg <= 0;
         accum_sum_b_reg <= 0;
         iter <= 0;
      end else begin

         accum_sum_a_reg <= accum_sum_a[pp_per_cycle];
         accum_sum_b_reg <= accum_sum_b[pp_per_cycle];
         iter <= iter + 1;
      end
endmodule
```

(b) Sketch the hardware that you expect to be synthesized for an $m = 2$ version. Make sure that your design does not do something foolish with the conventional adder.

The hardware appears below. Coloring has been used to emphasize the hardware corresponding to each iteration of the generate loop (blue and green) and hardware corresponding to Verilog outside of the generate loop (black). Pay close attention to `accum_sum_a[i]` and `accum_sum_b[i]`. They are declared outside the generate loop but are used to interconnect items in different generate loop iterations.

The diagram shows the inferred hardware, before any optimization. Note that the conventional adder (the big box with the plus) receives its inputs from the outputs of register `accum_sum_a_reg` and `accum_sum_b_reg`, rather than the CSA outputs. This gives the adder the entire clock period to produce its sum.



Problem 3: Run the synthesis program to compare the cost and performance of `mult_seq_csa_m` to `mult_seq_m`. The synthesis script `syn.cmd` can be used to synthesize these modules at different sizes. To run it use the command `rc -files syn.cmd`. Feel free to modify the script. (It is written in TCL, it should be easy to find information on this language.)

(a) Show the cost and performance versus m for these modules.

The cost and performance appear below. The first table shows the results using the unmodified synthesis script, in which area was minimized. The second table shows the results using a synthesis script in which the synthesis program was set to minimize delay.

```
--------------- Area Optimization ----------------------------------
Module Name                              Area   Clock    Total    Init.
                                              Period    Delay   Interv
mult_seq_csa_m_wid16_pp_per_cycle1     110308   14170   226720   226720
mult_seq_csa_m_wid16_pp_per_cycle2     135192   13692   109536   109536
mult_seq_csa_m_wid16_pp_per_cycle4     157668   12828    51312    51312
mult_seq_csa_m_wid16_pp_per_cycle8     195212   11110    22220    22220
mult_seq_m_wid16_pp_per_cycle1          74092   16444   263104   263104
mult_seq_m_wid16_pp_per_cycle2          99884   17470   139760   139760
mult_seq_m_wid16_pp_per_cycle4         112664   16508    66032    66032
mult_seq_m_wid16_pp_per_cycle8         154744   16463    32926    32926

--------------- Delay Optimization ---------------------------------
Module Name                              Area   Clock    Total    Init.
                                              Period    Delay   Interv
mult_seq_csa_m_wid16_pp_per_cycle1     164940    2054    32864    32864
mult_seq_csa_m_wid16_pp_per_cycle2     195408    2255    18040    18040
mult_seq_csa_m_wid16_pp_per_cycle4     239340    2756    11024    11024
mult_seq_csa_m_wid16_pp_per_cycle8     316748    4043     8086     8086
mult_seq_m_wid16_pp_per_cycle1         125408    3062    48992    48992
mult_seq_m_wid16_pp_per_cycle2         166488    3368    26944    26944
mult_seq_m_wid16_pp_per_cycle4         202096    3777    15108    15108
mult_seq_m_wid16_pp_per_cycle8         263772    4285     8570     8570
```

(b) If you solved the previous problem correctly the total delay shown for `mult_seq_csa_m` should be wrong. Explain why, and (optional) if you like try modifying syn.cmd to fix it.

The TCL script computes the total delay by multiplying the clock period by w/m. (In the TCL script w/m is computed by the routine `get_stages`. In that routine variable `bits` is used for w and `deg` for m.) The values of m and w used by the script are chosen so that m always divides w, so the problem has nothing to do with integer truncation errors.

The module designed for the solution to Problem 2 uses an extra cycle to compute the sum, so it takes $m/w + 1$ cycles, and the TCL script does not take this into account. (Of course, that would be easy enough to fix.)

(c) Explain how you might expect the total delay (time needed to compute a product) of `mult_seq_csa_m` to change with increasing m? Explain your expectation and whether the synthesis results bear that out.

The clock period is determined by either the delay of one carry-propagate (conventional) adder or the delay of m carry-save adders, whichever is larger. For small values of m the carry-propagate adder would have the larger delay. So, one might expect that the clock period for the modules with $m = 1$ and $m = 2$ would be the same. However, the time needed to compute a product, $T(m)$, would go from $T(1) = (w/1 + 1)\, t_{\rm clk} \approx w t_{\rm clk}$ to $T(2) = (w/2 + 1)\, t_{\rm clk} \approx \frac{w}{2} t_{\rm clk}$ which is nearly half the time. For these small values of m the clock period $t_{\rm clk} = t_{\rm latch} + t_{\rm adder}$, where $t_{\rm latch}$ is the setup time needed for the registers and $t_{\rm adder}$ is the time needed for the carry-propagate adder. When m is increased further the clock period time will be more like $t_{\rm clk} = t_{\rm latch} + m t_{\rm csa}$ where $t_{\rm csa}$ is the delay for one carry-save adder. At that point, further increases in m will not improve total performance by as much:

$$\begin{aligned} T(m) &= (w/m + 1)\,(t_{\rm latch} + m t_{\rm csa}) \\ &= (w+1) t_{\rm csa} + \left(\frac{w}{m} + 1\right) t_{\rm latch} \end{aligned}$$

When the synthesis program is optimizing delay, results are consistent with this analysis: Performance improvement with increasing m is much better when m is small than when m is large.

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2014 Homework 3
//

 /// SOLUTION

 /// Assignment http://www.ece.lsu.edu/koppel/v/2014/hw03.pdf
 /// Solution http://www.ece.lsu.edu/koppel/v/2014/hw03_sol.pdf

 /// Instructions:
  //
  // (1) Find the undergraduate workstation laboratory, room 126 EE
  //     Building.
  //
  // (2) Locate your account.  If you did not get an account please
  //     E-mail: koppel@ece.lsu.edu
  //
  // (3) Log in to a Linux workstation.
  //     The account should start up with a WIMP interface (windows, icons,
  //     mouse, pull-down menus)  ( :-) ) but one or two things need
  //     to be done from a command-line shell.  If you need to brush up
  //     on Unix commands follow http://www.ece.lsu.edu/koppel/v/4ltrwrd/.
  //
  // (4) If you haven't already, follow the account setup instructions here:
  //     http://www.ece.lsu.edu/koppel/v/proc.html
  //
  // (5) Copy this assignment, local path name
  //     /home/faculty/koppel/pub/ee4755/hw/2014f/hw03
  //     to a directory ~/hw02 in your class account. (~ is your home
  //     directory.) Use this file for your solution.
  //
  // (6) Find the problems in this file and solve them.
  //
  //     Your entire solution should be in this file.
  //
  //     Do not change module names.
  //
  // (7) Your solution will automatically be copied from your account by
  //     the TA-bot.


 /// Additional Resources
  //
  // Verilog Documentation
  //    The Verilog Standard
  //      http://standards.ieee.org/getieee/1800/download/1800-2012.pdf
  //    Introductory Treatment (Warning: Does not include SystemVerilog)
  //      Brown & Vranesic, Fundamentals of Digital Logic with Verilog, 3rd Ed.
  //
  // Account Setup and Emacs (Text Editor) Instructions
  //      http://www.ece.lsu.edu/koppel/v/proc.html
  //      To learn Emacs look for Emacs tutorial.
  //
  // Unix Help
  //      http://www.ece.lsu.edu/koppel/v/4ltrwrd/



////////////////////////////////////////////////////////////////////////////////
/// Behavioral Multiplier
```

```
module mult_behav_1
  #(int wid = 16)
   (output logic[2*wid-1:0] prod, input logic[wid-1:0] plier, cand);

   assign prod = plier * cand;
endmodule




////////////////////////////////////////////////////////////////////////////////
/// Simple m-Step Sequential Multiplier


module mult_seq_m #( int wid = 16, int pp_per_cycle = 2 )
   ( output logic [2*wid-1:0] prod,
     input logic [wid-1:0] plier,
     input logic [wid-1:0] cand,
     input clk);

   localparam int iterations = ( wid + pp_per_cycle - 1 ) / pp_per_cycle;
   localparam int iter_lg = $clog2(iterations);

   logic [iter_lg:1] iter;
   logic [2*wid-1:0] accum;

   // cadence translate_off
   initial iter = 0;
   // cadence translate_on

   always @( posedge clk ) begin

      if ( iter == iter_lg'(iterations) ) begin

         prod = accum;
         accum = 0;
         iter = 0;

      end

      for ( int i=0; i<pp_per_cycle; i++ )
        begin
           int pos;  pos = iter * pp_per_cycle + i;
           if ( cand[pos] ) accum += plier << pos;
        end

      iter++;

   end

endmodule


////////////////////////////////////////////////////////////////////////////////
/// An Sequential Multiplier using a Carry-Save Adder


// Examine this module for Problem 1.
// Don't modify the module.


`include "/apps/linux/cadence/RC141/share/synth/lib/chipware/sim/verilog/CW/CW_csa.v"
```

```
module mult_seq_csa #( int wid = 16 )
   ( output logic [2*wid-1:0] prod,
     input logic [wid-1:0] plier,
     input logic [wid-1:0] cand,
     input clk);

   localparam int wlog = $clog2(wid);

   logic [wlog-1:0] pos;

   logic [2*wid-1:0] accum_sum_a_reg, accum_sum_b_reg;
   wire              co;

   // cadence translate_off
   initial begin pos = 0; accum_sum_a_reg = 0; accum_sum_b_reg = 0;  end
   // cadence translate_on

   wire [2*wid-1:0] accum_sum_a, accum_sum_b;

   wire [2*wid-1:0] pp = cand[pos] ? plier << pos : 0;

   CW_csa #(2*wid) csa
     ( .carry(accum_sum_a), .sum(accum_sum_b), .co(co),
       .a(accum_sum_a_reg), .b(accum_sum_b_reg), .c(pp), .ci(1'b0) );

   always @( posedge clk ) pos <= pos + 1;

   always @( posedge clk ) begin

      if ( pos == wid-1 ) begin

         prod = accum_sum_a + accum_sum_b;
         accum_sum_a_reg = 0;
         accum_sum_b_reg = 0;

      end else begin

         accum_sum_a_reg = accum_sum_a;
         accum_sum_b_reg = accum_sum_b;

      end

   end

endmodule

////////////////////////////////////////////////////////////////////////////////
/// An m-bit Sequential Multiplier using a CSA


 /// Problem 2: Modify this module.


module mult_seq_csa_m #( int wid = 16, int pp_per_cycle = 2 )
   ( output logic [2*wid-1:0] prod,
     input logic [wid-1:0] plier,
     input logic [wid-1:0] cand,
     input clk);

   /// SOLUTION

   localparam int iterations = ( wid + pp_per_cycle - 1 ) / pp_per_cycle;
   localparam int iter_lg = $clog2(iterations);
   localparam int wid_lg = $clog2(wid);
```

```
   logic [iter_lg:0] iter;

   // cadence translate_off
   initial iter = 0;
   // cadence translate_on

   wire [2*wid-1:0] accum_sum_a[0:pp_per_cycle], accum_sum_b[0:pp_per_cycle];
   logic [2*wid-1:0] accum_sum_a_reg, accum_sum_b_reg;

   assign           accum_sum_a[0] = accum_sum_a_reg;
   assign           accum_sum_b[0] = accum_sum_b_reg;

   for ( genvar i=0; i<pp_per_cycle; i++ ) begin

      wire [wid_lg:1] pos = iter * pp_per_cycle + i;
      wire            co; // Unconnected.

      wire [2*wid-1:0] pp = pos < wid && cand[pos] ? plier << pos : 0;

      CW_csa #(2*wid) csa
        ( .sum(accum_sum_a[i+1]), .carry(accum_sum_b[i+1]), .co(co),
          .a(accum_sum_a[i]), .b(accum_sum_b[i]), .c(pp), .ci(1'b0) );

   end

   always @( posedge clk ) begin

      if ( iter == iterations ) begin

         // The commented-out line below shows the wrong way of
         // designing this module.
         //
         //  prod = accum_sum_a[pp_per_cycle] + accum_sum_b[pp_per_cycle];

         // Note that the product is computed by using the register
         // outputs, rather than the output of the last CSA.
         //
         prod <= accum_sum_a_reg + accum_sum_b_reg;

         accum_sum_a_reg <= 0;
         accum_sum_b_reg <= 0;
         iter <= 0;

      end else begin

         accum_sum_a_reg <= accum_sum_a[pp_per_cycle];
         accum_sum_b_reg <= accum_sum_b[pp_per_cycle];
         iter <= iter + 1;

      end

   end

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Pipelined Multiplier


module mult_pipe #( int wid = 16, int pp_per_stage = 2 )
   ( output logic [2*wid-1:0] prod,
```

```
      input logic [wid-1:0] plier,
      input logic [wid-1:0] cand,
      input clk);

   localparam int stages = ( wid + pp_per_stage - 1 ) / pp_per_stage;

   logic [2*wid-1:0] pl_accum[0:stages];
   logic [wid-1:0] pl_plier[0:stages];
   logic [wid-1:0] pl_cand[0:stages];

   always @( posedge clk ) begin

      pl_accum[0] = 0;
      pl_plier[0] = plier;
      pl_cand[0] = cand;

      for ( int stage=0; stage<stages; stage++ ) begin

         logic [2*wid-1:0] accum;  accum = pl_accum[stage];

         for ( int j=0; j<pp_per_stage; j++ ) begin

            int pos;  pos = stage * pp_per_stage + j;

            if ( pos < wid && pl_cand[stage][pos] )
              accum += pl_plier[stage] << pos;

         end

         pl_accum[stage+1] <= accum;
         pl_cand[stage+1] <= pl_cand[stage];
         pl_plier[stage+1] <= pl_plier[stage];

      end

   end

   assign prod = pl_accum[stages];

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Pipelined Multiplier, Instantiated Stages


module mult_pipe_stage #( int wid = 16, int pp_per_stage = 2, int stage = 0 )
   ( output logic [2*wid-1:0] accum_out,
     input [2*wid-1:0] accum_in,
     input [wid-1:0] plier,
     input [wid-1:0] cand);

   always @* begin

      logic [2*wid-1:0] accum;  accum = accum_in;

      for ( int j=0; j<pp_per_stage; j++ ) begin

            int pos;  pos = stage * pp_per_stage + j;

            if ( pos < wid && cand[pos] ) accum += plier << pos;

      end
```

```
      accum_out = accum;

   end

endmodule

module mult_pipe_ia #( int wid = 16, int pp_per_stage = 2 )
   ( output logic [2*wid-1:0] prod,
     input logic [wid-1:0] plier,
     input logic [wid-1:0] cand,
     input clk);

   localparam int stages = ( wid + pp_per_stage - 1 ) / pp_per_stage;

   logic [2*wid-1:0] pl_accum[0:stages];
   logic [wid-1:0] pl_plier[0:stages];
   logic [wid-1:0] pl_cand[0:stages];

   always @* begin

      pl_accum[0] = 0;
      pl_plier[0] = plier;
      pl_cand[0] = cand;

   end

   for ( genvar stage = 0;  stage < stages;  stage++ ) begin

      wire logic [2*wid-1:0] accum;

      mult_pipe_stage_x #(wid, pp_per_stage, stage) this_stage
        ( accum, pl_accum[stage], pl_plier[stage], pl_cand[stage]);

      always @( posedge clk ) begin
         pl_accum[stage+1] <= accum;
         pl_plier[stage+1] <= pl_plier[stage];
         pl_cand[stage+1] <= pl_cand[stage];
      end

   end

   assign prod = pl_accum[stages];

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Testbench Code

// cadence translate_off

module testbench;

   localparam int wid = 16;
   localparam int num_tests = 1000;
   localparam int NUM_MULT = 10;
   localparam int err_limit = 7;
   localparam bit pipeline_test_exact = 1;

   logic clock;

   always #1 clock <= !clock;

   logic [wid-1:0] plier, cand;
```

```
    logic [wid-1:0] plierp, candp;
    logic [2*wid-1:0] prod[NUM_MULT];
    logic [2*wid-1:0] prodp[NUM_MULT];

    mult_behav_1 #(wid) mb1(prod[0], plier, cand);

    mult_seq_m   #(wid,8) ms44(prod[1], plier, cand, clock);
    mult_seq_m   #(wid,3) ms43(prod[2], plier, cand, clock);
    mult_seq_csa   #(wid) mc(prod[3], plier, cand, clock);
    mult_seq_csa_m   #(wid,4) mc4(prod[4], plier, cand, clock);
    mult_seq_csa_m   #(wid,1) mc1(prod[5], plier, cand, clock);

    localparam int ppps_2 = 1;

    mult_pipe    #(wid,4) mp4(prodp[6], plierp, candp, clock);
    mult_pipe    #(wid,ppps_2) mp3(prodp[7], plierp, candp, clock);
    mult_pipe_ia #(wid,4) mpi4(prodp[8], plierp, candp, clock);
    mult_pipe_ia #(wid,ppps_2) mpi3(prodp[9], plierp, candp, clock);

    string names[] = '{"Behav_1",
                       "Seq m4",
                       "Seq m3",
                       "Seq CSA",
                       "Seq CSA m4",
                       "Seq CSA m1",
                       "Pipelined m4",
                       "Pipelined m1",
                       "Pipelined IA m4",
                       "Pipelined IA m1"
                       };

    int err_cnt[NUM_MULT];

    // Array of multiplier/multiplicand values to try out.
    // After these values are used a random number generator will be used.
    //
    int tests[$] = {1,1, 1,2,  2,1, 'h10,'h20, 1,32,  32, 1};

    initial begin

       clock = 0;

       for ( int i=0; i<num_tests; i++ ) begin

          // Change input to pipelined units.
          //
          for ( int t=0; t<=wid; t++ ) begin
             plierp = t;
             candp = 256;
             #2;
          end

          // Set multiplier and multiplicand values for non-piped units.
          //
          plier = tests.size() ? tests.pop_front() : $random();
          cand = tests.size() ? tests.pop_front() : $random();

          // Set multiplier and multiplicand values for piped units.
          //
          plierp = plier;
          candp = cand;

          // For pipelined units, copy output at the time it should be ready.
          //
          fork
```

```
            #(2 * wid/4) prod[6] = prodp[8];
            #(2 * wid/4) prod[8] = prodp[8];
            #(2 * ((wid+ppps_2-1)/ppps_2)) prod[7] = prodp[7];
            #(2 * ((wid+ppps_2-1)/ppps_2)) prod[9] = prodp[9];
         join_none

         if ( pipeline_test_exact ) begin

            // Modify the inputs to the pipelined units in subsequent cycles.
            //
            for ( int t=0; t<=wid; t++ ) begin
               #2;
               plierp = t;
               candp = 1;
            end

            plierp = 0;
            candp = 0;

         end

         #1000;

         // Make sure each module's output is correct.
         //
         for ( int mut=1; mut<NUM_MULT; mut++ ) begin

            if ( prod[0] !== prod[mut] ) begin

               err_cnt[mut]++;

               if ( err_cnt[mut] < err_limit )
                 $display("Error in %s test %4d:  %x != %x (correct)\n",
                          names[mut], i, prod[mut], prod[0]);
            end

         end

      end

      // Tests completed, report error count for each device.
      //
      for ( int mut=1; mut<NUM_MULT; mut++ ) begin

         $display("Mut %s, %d errors (%.1f%% of tests)\n",
                  names[mut], err_cnt[mut],
                  100.0 * err_cnt[mut]/real'(num_tests) );

      end

      $finish(2);

   end

endmodule

// cadence translate_on
```

LSU EE 4755 Homework 4 Solution Due: 24 November 2014

Problem 0: Copy the code package from `/home/faculty/koppel/pub/ee4755/hw/2014f/hw04`. Verify that everything is working by running the simulation on the unmodified file. It should report that there is correct output but no compression:

```
Correct output, strings match. But no compression!
In size         117 bytes, out size         117 bytes.
```

Problem 1: Module `asc_to_bin` is to filter a stream of ASCII characters so that ASCII decimal numbers are replaced by binary numbers preceded by an escape character. The idea is to reduce the size of data streams that contain lots of large numbers. For example, consider the sentence, "There are 31536000 seconds in a year." The module `asc_to_bin` should replace that sequence of eight ASCII characters 31536000 with an escape character and an integer encoding of the number.

The module has an 8-bit input and output for the character, `char_in` and `char_out`. There is a 1-bit input `can_insert` which is true when the module can read a character from `char_in`. If input `insert_req` is asserted when `can_insert` is true then the character on `char_in` will be read.

There is a 1-bit output `can_remove` which is true when the character on `char_out` is valid. (It would not be valid if the module does not contain any characters and for other reasons.) If input `remove_req` is set to `1` and `can_remove` is true then the character at `char_out` will change to the next character or, if that's the last available character, `can_remove` will go to zero.

There is also a 1-bit input `reset`. If `reset` is high at the positive edge of the clock then the module should reset itself.

Initially in the homework package, module `asc_to_bin` passes through characters unchanged. Modify it so that it converts ASCII decimal numbers to binary as described above.

At the end of the simulation the testbench will indicate whether the output string is correct, and the original and compressed sizes. For example, the output using the unmodified code package will be:

```
Correct output, strings match. But no compression!
In size         117 bytes, out size         117 bytes.
```

The testbench also provides a trace showing some information each time a character is removed. For the unmodified code,

```
ncsim> run
c  79 = O   tail  1 head  0
c 110 = n   tail  3 head  1
c 101 = e   tail  4 head  2
c  32 =     tail  7 head  3
c  49 = 1   tail  8 head  4
```

The character removed is shown as a decimal number and as a character, for example 110 and "n" for the second line. Also shown are the values of two objects in the `asc_to_int` module, `tail` and `head`. Feel free to add your own variables to the list. Search for "Trace execution" to find the code that prints this trace.

The parameter `max_chars` indicates the maximum size of the integer that should be created. Currently the testbench expects all integers to be of this size.

Keep the following in mind:

- Do not convert a number to binary if it would take more space than the original.
- The module must be synthesizable.
- The synthesized hardware must be reasonably efficient.

For extra credit, modify both the `asc_to_bin` module and the testbench so that `asc_to_bin` can compress a string of ASCII digits to the smallest integer (in multiple of bytes) that can hold the integer. (The current behavior is to use one size integer, determined by parameter `max_chars`.)

The complete solution can be found at `/home/faculty/koppel/pub/ee4755/hw/2014f/hw04/hw04-sol.v` and on the Web at `http://www.ece.lsu.edu/koppel/v/2014/hw04-sol.v.html`.

Encoding the incoming ASCII characters as an integer is straightforward. The tricky part is sending the encoded integer and escape character to the module outputs at the correct time. Remember that characters are removed from the module only when the external device requests them (asserts `remove_req`) so one can't assume that something that has just been read can immediately be sent to the output.

The following approach is used in the solution. Two registers hold the encoded binary values. The design encodes incoming digits as they arrive into register `val_encode`, a second register `val_wait`, holds completed integers that are worth using (the ASCII version is not too short).

Let's suppose the value of `tail` was 7 when the first ASCII digit of a suitable string of digits arrived. Normally, the first ASCII character of this string would be sent to the module output when `head` reaches 7. What we want instead is that the escape character be sent to the output, followed by the bytes of the binary number in subsequent cycles. The solution uses a new array, `esc_here`, to indicate that an encoded integer starts here. If `esc_here[head]` is 1 the module will output an escape character and will switch to using `val_wait` as the source of module outputs for the next `max_chars` characters. It then returns to using `storage` as the source of characters.

Array `esc_here[tail]` is set to zero each time a character is read. When the start of a string of digits is detected (see `start_encoding`) the tail location is saved in `tail_at_enc_start`. When encoding is to end (either due to a non-digit character or overflow, see `end_encoding`) if the encoded number can be used (ASCII number not too short, and `val_wait` is not being used, see `use_encoding`) then we set `esc_here[tail_at_enc_start]=1`.

The updating of the `head` pointer has not been modified: it's incremented at each `remove_req`. However `tail` is adjusted whenever an encoded value is to be used. If the encoding reduces the number of characters by x then x is subtracted from `tail`.

An alternative to using `val_wait` would be to write the escape character and encoded integer into storage. This would simplify the design by removing the multiplexor at the character output and the associated "drain" logic (see the solution code), but it would require a second write port for storage.

Problem 2: Synthesize your module.

(*a*) Indicate the cost and performance with and without timing optimization. (With timing optimization means using `define_clock`.)

See the table below. The column headed "Timing Constr" indicates the kind of timing optimization. *None* means that no timing constraints were specified and so there was no timing optimization. *Reg -¿ Reg* means that the Encounter `define_clock` command was used, and so timing was optimized from register outputs to register inputs. However the timing of paths starting at module inputs or leading to module outputs was ignored. For the column headed *In, Reg -¿ Reg, Out* the `define_clock` command was used and `external_delay` was also used to indicate the assumption that module inputs are available at the beginning of the clock cycle and that module outputs are expected to be available at the end of the clock cycle.

Without timing optimization the module is 20% cheaper but five times slower. The optimizations were performed with effort set to medium.

Module Name	Area	Clock Period	Timing Constr
asc_to_bin_sol 4 4	206728	20084	None
asc_to_bin_sol 4 4	255460	3844	Reg -> Reg
asc_to_bin_sol 4 4	251736	3687	In, Reg -> Reg, Out

(*b*) Even if `define_clock` is used, the synthesis program won't optimize all paths, only those with both ends affected by the clock. Show how to use the Encounter `external_delay` command to get the proper timing optimization.

The first command below tells Encounter that all inputs are assumed to be available at the beginning of the clock period for `my_clk` (which needs to have been defined with `define_clock`). The second tells encounter that all outputs are expected to be stable at the end of the clock period. The stuff in the square brackets returns the list of ports, the port names could also have been typed by hand.

```
external_delay -clock my_clk -output 0 [find /designs/*/ports_out/ -port *]
external_delay -clock my_clk -input 0 [find /designs/*/ports_in/ -port *]
```

```
////////////////////////////////////////////////////////////////////////////////
//
/// LSU EE 4755 Fall 2014 Homework 4
//
 /// SOLUTION

 // Assignment  http://www.ece.lsu.edu/koppel/v/2014f/hw04.pdf

 /// The solution is in module asc_to_bin_sol.


typedef enum { Char_escape = 1, Char_0 = 48, Char_9 = 57 } Chars_Special;

module asc_to_bin
  #( int size_lg = 4,
     int max_chars = 4,
     int size = 1 << size_lg )
  ( output [7:0] char_out,
    output can_insert, can_remove,
    input [7:0] char_in,
    input insert_req, remove_req,
    input reset, clk);

   logic [7:0] storage [size];

   logic [size_lg:1] head, tail;

   uwire   is_digit = char_in >= Char_0 && char_in <= Char_9;

   uwire   empty = head == tail;
   uwire   full = tail + 1 == head;

   assign can_insert = !full;
   assign can_remove = !empty;

   assign char_out = storage[head];

   // cadence translate_off
   initial for ( int i=0; i<size; i++ ) storage[i] = 255;
   // cadence translate_on

   always @( posedge clk ) if ( reset ) begin

      tail <= 0;

   end else begin

      if ( insert_req ) begin

         storage[tail] = char_in;
         tail <= tail + 1;

      end

   end

   always @( posedge clk ) if ( reset ) begin

      head <= 0;

   end else if ( remove_req ) begin

      head <= head + 1;

   end

endmodule


module asc_to_bin_sol
  #( int size_lg = 4,
     int max_chars = 4,
     int size = 1 << size_lg )
  ( output [7:0] char_out,
    output can_insert, can_remove,
    input [7:0] char_in,
    input insert_req, remove_req,
    input reset, clk);

   // Storage for characters.
   logic [7:0] storage [size];

   // Location at which encoded number should start.  That is,
   // if esc_here[x] is 1, then storage[x] is the first character of
   // an ASCII string that should be replaced with an escape character
   // and a binary encoded value;
   logic esc_here [size];

   // Register used for preparing encoded integer.
   logic [max_chars-1:0][7:0] val_encode;
   // Register for holding encoded integer until all characters removed.
   logic [max_chars-1:0][7:0] val_wait;

   // True if val_wait holds a value that has not yet been read.
   logic val_wait_full;
```

```
// Pointers into storage.
logic [size_lg:1] head;      // Location being read (sent to module output).
uwire [size_lg:1] write_idx;  // Next location to write.
logic [size_lg:1] tail;      // Possible next location to write.
logic [size_lg:1] tail_at_enc_start;

// Note: encoding refers to the process of converting a string of
// ASCII characters to an integer.
uwire now_encoding, end_encoding;
logic was_encoding;

logic [7:0] ascii_int_len;

// Note: draining refers to sending the bytes in val_wait to the
// module outputs.
logic draining;
logic [$clog2(max_chars)-1:0] drain_idx;
uwire   start_draining, end_draining;

uwire   empty, full;

// cadence translate_off
initial for ( int i=0; i<size; i++ ) storage[i] = 255;
// cadence translate_on


///
/// Hardware For Encoding ASCII Digits into Binary
///

// Check whether a digit is present.
//
uwire is_digit = char_in >= Char_0 && char_in <= Char_9;
uwire is_nz_digit = char_in > Char_0 && char_in <= Char_9;  // Non-Zero

// Convert ASCII digit to an integer.
//
uwire [3:0] char_bin = char_in - Char_0;

// Combine digit at char_in with current value of val_encode.
//
uwire [max_chars:0] [7:0] next_val_encode = val_encode * 10 + char_bin;
uwire overflow = next_val_encode[max_chars] != 0;

///
/// Hardware to Detect the Start, End, and Suitability of a String of Digits
//

logic was_digit;
always @( posedge clk )
  if ( reset )           was_digit <= 0;
  else if ( insert_req ) was_digit <= is_digit;

// True if we should start encoding a string of digits.
uwire start_encoding =
     insert_req && is_nz_digit && ( !was_digit || overflow );

assign now_encoding = start_encoding || was_encoding && !end_encoding;
always @( posedge clk )
  if ( reset )           was_encoding <= 0;
  else if ( insert_req ) was_encoding <= now_encoding;

// True if encoding should end, whether or not the encoding will be used.
assign end_encoding =
       insert_req && was_encoding && ( !is_digit || overflow );

// True if encoded integer should be used.
// We don't want to do this if the ASCII string is too short,
// or if val_wait is still occupied.
uwire use_encoding = end_encoding
     && ( ascii_int_len > max_chars )
     && ( !val_wait_full || end_draining );


// Update registers holding encoded integer, and those keeping
// track of locations.
//
always @( posedge clk ) if ( insert_req ) begin

   if ( start_encoding ) begin

      // Initialize val_encode with first character.
      val_encode <= char_bin;

      // Remember where ASCII digits started.
      tail_at_enc_start <= write_idx;

      // Keep track of how many digits there are.
      ascii_int_len <= 1;

   end else begin

      // Update registers assuming that we are continuing to
      // encode. (It doesn't hurt if we are not currently encoding.)
      val_encode <= next_val_encode;
      ascii_int_len <= ascii_int_len + 1;
```

```
   end

   // Move val_encode to a second register so that the next string of
   // ASCII digits can be encoded without having to wait for this
   // value to be removed.
   if ( use_encoding ) val_wait <= val_encode;

end


///
/// Hardware for Writing Characters into Storage
///

// If the encoded integer is used we need to move the tail back by
// the number of characters saved. That's easier to compute using
// the location at which the encoded number started.
uwire [size_lg:1] tail_adj = tail_at_enc_start + max_chars + 1;

// Location at which to write current character.
assign write_idx = use_encoding ? tail_adj : tail;


/// Write the Storage and the Tail Pointer
//
always @( posedge clk ) if ( reset ) begin

   tail <= 0;

end else if ( insert_req ) begin

   // We've decided to use an encoded number. Remember where.
   // When head reaches tail_at_enc_start we will start sending
   // the encoded number to the output.
   if ( use_encoding ) esc_here[tail_at_enc_start] <= 1;

   storage[write_idx] <= char_in;
   esc_here[write_idx] <= 0;

   tail <= write_idx + 1;

end


///
/// Hardware For Removing Characters From Storage
///

/// Character Out Mux
//
//  The char_out port can be connected to three things:
//
//    - A memory holding stored characters: storage[].
//    - The escape character (a constant, Char_escape).
//    - A register holding a number encoded in binary, val_wait.
//
assign char_out =
       start_draining ? Char_escape :
       draining       ? val_wait[drain_idx]
                      : storage[head];

assign start_draining = !empty && esc_here[head];
assign end_draining = remove_req && draining && drain_idx == 0;

// Update the register that indicates whether val_wait is holding
// something.
always @( posedge clk )
  if ( reset )              val_wait_full <= 0;
  else if ( use_encoding ) val_wait_full <= 1;
  else if ( end_draining ) val_wait_full <= 0;

always @( posedge clk ) if ( reset ) begin

   draining <= 0;
   drain_idx <= 0;
   head <= 0;

end else if ( remove_req ) begin

   draining <= start_draining ? 1 : drain_idx == 0 ? 0 : draining;

   drain_idx <= start_draining ? max_chars-1
              : drain_idx > 0 ? drain_idx - 1 : 0;

   head <= head + 1;

end


///
/// Hardware Related to Storage Full and Empty Status
///

assign empty = head == tail;
assign full = tail + 1 == head;

assign can_remove = !empty && ( !now_encoding || head != tail_at_enc_start );
assign can_insert = !full;
```

```
endmodule



// cadence translate_off
module testbench();

   localparam int elts_lg = 4;
   localparam int elts = 1 << elts_lg;
   localparam int int_chars = 2;

   uwire [7:0] char_out;
   uwire       can_insert, can_remove;
   logic [7:0] char_in;
   logic insert_req, remove_req, reset, clk;
   asc_to_bin_sol #(elts_lg,int_chars) b1
     (char_out,can_insert,can_remove,char_in,insert_req,remove_req,reset,clk);

   int cycle_num;

   initial begin
      clk = 0;
      cycle_num = 0;
      fork
         forever #1 clk = !clk;
         forever @( posedge clk ) cycle_num++;
      join
   end

   string in_str = "One 1 two 12 three 317 four 1029 six 123456 ten 1234567890.  There are 60 seconds in a minute and 31536000 in a year."
   string out_str = "";

   initial begin

      automatic int insert_finished_cyc = 0;
      automatic int out_size = 0;
      automatic bit tb_insert_done = 0;
      automatic bit tb_remove_done = 0;

      /// Reset the module.
      //
      reset = 0;
      insert_req = 0;
      remove_req = 0;
      @( negedge clk ) reset = 1;
      @( negedge clk ) reset = 0;
      @( negedge clk );

      /// Check for one possible error.
      //
      if ( can_insert !== 1 ) begin

         $display("Module did not reset, can_insert: %h\n", can_insert);
         $fatal(1);

      end

      /// Start Main Testing Loops
      //
      fork

         /// Watchdog -- Stop simulation if it's taking too long.
         //
         fork begin

            automatic int cyc_limit = in_str.len() * 100;

            fork
               wait ( cycle_num == cyc_limit );
               wait ( tb_insert_done && tb_remove_done );
            join_any

            if ( cycle_num >= cyc_limit ) begin
               $display("Exceeded cycle limit, exiting.\n");
               $fatal(1);
            end

         end join_none

         /// Trace Execution -- Print Signal Values After Interesting Changes
         //
         while ( !tb_insert_done || !tb_remove_done ) begin

            @( insert_req or remove_req or can_insert or can_remove
               or b1.tail or b1.head or tb_insert_done or tb_remove_done );
            @( negedge clk );

            /// Trace execution by showing removed character and
            /// related information.
            //
            $display( "c In %c  Out %d = %c   tail %d head %d  b2 %d",
                      char_in, char_out, char_out, b1.tail, b1.head, b1.val_wait);

         end
```

```
      /// Insert Characters
      //
      begin

         automatic int in_pos = 0;

         while ( in_pos < in_str.len() ) begin

            @( negedge clk );

            // Flip a coin, and if it comes up tails send a character
            // in if module is ready for one.
            //
            if ( {$random} & 'h1 && can_insert ) begin

               char_in = in_str[in_pos++];
               insert_req = 1;

            end else begin

               insert_req = 0;

            end

         end

         @( negedge clk );

         insert_req = 0;
         insert_finished_cyc = cycle_num;

         $display("Done feeding inputs.");
         tb_insert_done = 1;

      end

      /// Remove Characters
      //
      begin

         int buffer;
         automatic int bytes_remaining = 0;

         while ( insert_finished_cyc == 0
                 || cycle_num < insert_finished_cyc + elts * 10 ) begin

            @( negedge clk );

            if ( {$random} & 1 && can_remove ) begin

               remove_req = 1;
               out_size++;

               if ( bytes_remaining > 0 ) begin

                  buffer = ( buffer << 8 ) + char_out;
                  bytes_remaining--;

                  if ( bytes_remaining == 0 ) begin

                     // Convert binary number back to ASCII.
                     string iasc;
                     iasc.itoa(buffer);
                     out_str = {out_str,iasc};

                  end

               end else if ( char_out == Char_escape ) begin

                  bytes_remaining = int_chars;
                  buffer = 0;

               end else begin

                  out_str = {out_str,char_out};

               end

            end else begin

               remove_req = 0;

            end

         end

         $display("Done gathering outputs.\n");
         tb_remove_done = 1;

      end

   join

   if ( in_str != out_str )
     $display("** Error - strings don't match.\n");
   else
     $display("Correct output, strings match. %s",
              ( in_str.len() == out_size ) ? "But no compression!" : "");
```

```
      $display("In size %d bytes, out size %d bytes.\n",
               in_str.len(), out_size);
      $display("In - %s\nOut- %s\n",
               in_str, out_str);

      $finish(2);

   end

endmodule

// cadence translate_on
```

25 Spring 2001 Solutions

/// Solution to LSU EE 4702-1 Spring 2001 Homework 1

```
module priority_encoder_1_b(grant,found_out,request,found_in);
   output grant, found_out;
   input  request, found_in;

   wire   request, found_in;
   reg    grant, found_out;

   always @( request or found_in ) begin
      found_out =  found_in | request;
      grant     = !found_in & request;
   end

endmodule

module priority_encoder_1_es(grant,found_out,request,found_in);
   output grant, found_out;
   input  request, found_in;

   wire   grant, found_out;
   wire   request, found_in;

   or o1(found_out, found_in, request);
   and a1(grant, not_found_in, request);
   not n1(not_found_in, found_in);

endmodule

module priority_encoder_1_is(grant, found_out, request, found_in);
   input request, found_in;
   output grant, found_out;

   wire   found_out = found_in | request;
   wire   grant     = !found_in & request;

endmodule

module test_pe(done, okay_b, okay_is, okay_es);
   output done, okay_b, okay_is, okay_es;

   reg    done, okay_b, okay_is, okay_es;
   reg    request, found_in;
   wire   grant_b, found_out_b;
   wire   grant_is, found_out_is;
   wire   grant_es, found_out_es;

   priority_encoder_1_b peb(grant_b,found_out_b,request,found_in);
   priority_encoder_1_es pees(grant_es,found_out_es,request,found_in);
   priority_encoder_1_is peis(grant_is,found_out_is,request,found_in);

   reg [1:0]  answers [0:3];
   integer    i;

   initial begin

      done    = 0;
      okay_b  = 1;  okay_es = 1;  okay_is = 1;

      answers[2'b00] = 2'b00;
      answers[2'b01] = 2'b01;
      answers[2'b10] = 2'b11;
```

```
      answers[2'b11] = 2'b01;

      for(i=0; i<4; i=i+1) begin

         {request, found_in } = i;

         #1;

         if( {grant_b,found_out_b}   !== answers[i] ) okay_b  = 0;
         if( {grant_is,found_out_is} !== answers[i] ) okay_is = 0;
         if( {grant_es,found_out_es} !== answers[i] ) okay_es = 0;

      end // for (i=0; i<4; i=i+1)

      done = 1;

   end // initial begin

endmodule // test_pe
```

```
////////////////////////////////////////////////////////////////////////////////
///
/// Solution to LSU EE 4702-1 Spring 2001 Homework 3
///
///
//  Includes a testbench (which was not graded).


`timescale 1us/1us

module microwave_oven_controller(beep,dmt,dmu,dst,dsu,mag_on,key_code,clk);
   input key_code;              // Key begin pressed (see parameters).
   input clk;                   // A 64 Hz clock.
   output mag_on;               // When 1, magnetron is on (oven is heating).
   output beep;                 // When 1, emit tone.
   output dmt;                  // Tens digit of minute display.
   output dmu;                  // Units digit of minute display.
   output dst, dsu;             // Tens and units digits of seconds display.

   wire   clk;
   wire [5:0] key_code;
   reg [3:0]  dmt, dmu, dst, dsu;
   reg        mag_on;

   parameter  key_none  = 6'd0;  // No key pressed.
   parameter  key_never = 6'd1;  // This code will never be returned.
   parameter  key_start = 6'd10;
   parameter  key_reset = 6'd11;
   parameter  key_power = 6'd12;

   parameter  key_0 = 6'd20;
   parameter  key_1 = 6'd21;
   parameter  key_2 = 6'd22;
   parameter  key_3 = 6'd23;
   parameter  key_4 = 6'd24;
   parameter  key_5 = 6'd25;
   parameter  key_6 = 6'd26;
   parameter  key_7 = 6'd27;
   parameter  key_8 = 6'd28;
   parameter  key_9 = 6'd29;

   /// States
   //
   parameter  st_reset    = 0;
   parameter  st_entry_1  = 1;  // One digit entered.
   parameter  st_entry_1p = 2;  // One digit and power.
   parameter  st_entry_n  = 3;  // At least 2 digits (including power level).
   parameter  st_heating  = 4;
   parameter  st_paused   = 5;

   reg [3:0]  state, next_state;

   reg [2:0]  digit_count;      // Number of digits entered.
   reg [3:0]  power;            // Power level set by user.

   reg [5:0]  key_type;         // Type of key. (unless digit, key_code)
   parameter  kty_digit = 6'd30;

   // Number of tics before beep stops.  Zero if not beeping.
   //
   reg [7:0]  beep_timer;
   assign     beep = | beep_timer;
   always @( posedge clk ) if( beep_timer ) beep_timer = beep_timer - 1;
```

```
// Add Digit to Display
//
task add_digit;
   begin
      dmt = dmu;
      dmu = dst;
      dst = dsu;
      dsu = key_code - key_0;
      digit_count = digit_count + 1;
   end
endtask // add_digit

// Actions when switching to st_reset, including setting next_state.
//
task do_reset;
   begin
      dmt = 0;
      dmu = 0;
      dst = 0;
      dsu = 0;
      digit_count = 0;
      beep_timer = 0;
      next_state = st_reset;
      mag_on = 0;
   end
endtask // do_reset

initial begin do_reset; state = st_reset; end

/// State Transitions
//
always @( key_code ) if( key_code != key_none ) begin

   key_type = key_code >= key_0 && key_code <= key_9
              ? kty_digit : key_code;

   casez( {state, key_type} )

     {st_reset,kty_digit}:
       begin
          add_digit;
          power = 10;
          next_state = st_entry_1;
       end

     {st_entry_1p,kty_digit}:
       begin
          dsu = 0;
          add_digit;
          next_state = st_entry_n;
       end

     {st_entry_n,kty_digit}:
       begin
          if( digit_count == 4 )
            beep_timer = 16;
          else
            add_digit;
          next_state = state;
       end

     {st_entry_1,kty_digit}:
       begin
```

```
            add_digit;
            next_state = st_entry_n;
          end

        {st_entry_1,key_power}:
          begin
            power = dsu;
            next_state = st_entry_1p;
          end

        {st_entry_n,key_start}, {st_entry_1,key_start}:
          begin
            if( dst > 5 )
              begin
                beep_timer = 16;
                next_state = state;
              end else
                next_state = st_heating;
          end

        {st_heating,key_reset}:
          begin
            disable HEAT_LOOP;
            next_state = st_paused;
          end

        {st_paused,key_start}:
          begin
            next_state = st_heating;
          end

        {4'b????,key_reset}: do_reset;

        default: beep_timer = 16;


      endcase // casez( {state, key_type} )

      state = next_state;

  end // if ( key_code != key_none )

  // Clock Divider
  //
  // Divides 64 Hz clock by 64 so that sec_timer == 0 once per second.
  //
  reg [5:0]  sec_timer;
  initial sec_timer = 0;
  always @( posedge clk ) sec_timer = sec_timer + 1;



  always wait( state == st_heating ) begin
     fork:HEAT_LOOP
       reg [7:0]  on_timer, off_timer;

        // Turn magnetron on and off.
        //
       forever begin
          on_timer  = power * 64 * 2.5 / 10;
          off_timer = 64 * 2.5 - on_timer;
          mag_on    = 1;
          while ( on_timer ) @( posedge clk ) on_timer = on_timer - 1;
          if( off_timer ) begin
             mag_on = 0;
```

```
            while ( off_timer ) @( posedge clk ) off_timer = off_timer - 1;
          end
        end

        // Update display during heating.
        //
        forever @( posedge | sec_timer ) begin:T

           if( {dmt,dmu,dst,dsu} == 0 ) begin
              beep_timer = 128;
              state = st_reset;
              mag_on <= 0;
              disable HEAT_LOOP;
           end

           if( dsu ) begin dsu = dsu - 1; disable T; end
           dsu = 9;
           if( dst ) begin dst = dst - 1; disable T; end
           dst = 5;
           if( dmu ) begin dmu = dmu - 1; disable T; end
           dmu = 9;
           if( dmt ) dmt = dmt - 1;

        end // block: T
      join
      mag_on = 0;
   end // always wait

endmodule // microwave_oven_controller



module test_oven();

   reg  clk;
   wire [3:0] dmt, dmu, dst, dsu;
   wire       mag_on;
   reg [5:0]  key_mod;
   reg        reset;

   // Set this to 1 to have each change in the oven display appear
   // on the console.
   reg        monitor_display;
   // Set this to one to have each key press appear on the console.
   reg        monitor_keys;
   // Set this to one to get long test.
   reg        patient;

   microwave_oven_controller oven(beep,dmt,dmu,dst,dsu,mag_on,key_mod,clk);

   time       tics;

   initial tics = 0;
   always begin clk = 0; #5625; tics = tics + 1; #0; clk=1; #10000; end

   parameter   ss_reset  = 3'd0;
   parameter   ss_digit1 = 3'd1; // Single digit, power not entered.
   parameter   ss_digit2 = 3'd2; // Power entered or > 1 digit.
   parameter   ss_cook   = 3'd3;
   parameter   ss_pause  = 3'd4;

   parameter   kty_digit = 6'd30;

   reg [15:0]  shadow_display, alt_display;
   reg [2:0] shadow_state;
```

```
   integer   shadow_secs, mod_secs, delta;
   integer   shadow_tics, pause_tics, start_tics;
   integer   shadow_power, shadow_digits;
   integer   expected_beep_done, expecting_done_beep;

   integer   watch_display;

   integer   error_display, error_mag, error_beep, error_total;

`include "oven_keys.v"

   function integer abs;
      input a;
      integer a;
      abs = a < 0 ? -a : a;
   endfunction // abs


`define check_digit(d,l)                                                \
   if( (d) > (l) || (d) < 0 ) begin                                     \
      error_display = error_display + 1;                                \
      secs = -1;                                                        \
      disable tosecs;                                                   \
                     end

   // Convert time on display to seconds.
   //
   task tosecs;
      output secs;
      integer secs;

      begin
         `check_digit(dmt,9)
         `check_digit(dmu,9)
         `check_digit(dst,5)
         `check_digit(dsu,9)
         secs = dmt * 600 + dmu * 60 + dst * 10 + dsu;
      end

   endtask // tosecs

   /// Listen Beep
   //
   always @( beep )
     if( beep ) begin
        if( expecting_done_beep )
          begin
             expected_beep_done = 128;
             expecting_done_beep = 0;
          end
        if( expected_beep_done == 0 ) begin
           $display("Should not be beeping.");
           error_beep = error_beep + 1;
        end
     end else begin:B // if ( beep )
        integer delta;
        delta = abs( tics - expected_beep_done );
        if( shadow_state != ss_reset
            && expected_beep_done && delta > 5 ) begin
           $display("Beep wrong time. %d",delta);
           error_beep = error_beep + 1;
        end
        expected_beep_done = 0;
     end
```

```
/// Watch Magnetron
//
integer mag_on_start, mag_on_total;

always @( mag_on )
  if( mag_on ) begin
     if( shadow_state != ss_cook )
       begin
          $display("Mag on when cooking off.");
          error_mag = error_mag + 1;
       end
     if( mag_on_start != 0 ) begin:A
        integer cycle_this;
        cycle_this = tics - mag_on_start;
        if( shadow_power > 0 && shadow_power < 10
            && abs( cycle_this - 160 ) > 10 )
          begin
             $display("Mag cycle error.");
             error_mag = error_mag + 1;
          end
     end
     mag_on_start = tics;
  end else begin // if ( mag_on )
     mag_on_total = mag_on_total + tics - mag_on_start;
  end // else: !if( mag_on )

// Verify correct magnetron-on time.
//
task verify_cooking;
   begin:A
      integer correct_mag_tics;
      integer delta;

      correct_mag_tics = shadow_tics * shadow_power / 10;

      delta = abs(correct_mag_tics - mag_on_total);

      if( delta > 128 ) begin
         $display("Wrong power level. %d %d ",
                  correct_mag_tics, mag_on_total);
         error_mag = error_mag + 1;
      end
   end // block: A
endtask // verify_cooking

///  Watch display, etc.
//
always @( dmt or dmu or dst or dsu or shadow_display ) #1 begin

   if( monitor_display ) $display("Display: %d%d:%d%d",dmt,dmu,dst,dsu);

   if( shadow_state == ss_cook ) begin

      shadow_secs = shadow_secs - 1;
      shadow_tics = shadow_tics + 64;

      tosecs(mod_secs);

      if( mod_secs !== shadow_secs )
        begin
           $display("Wrong count. (cooking)");
           error_display = error_display + 1;
        end
```

```
      delta = shadow_tics - ( tics - start_tics );
      if( abs(delta) > 96 )
        begin
           $display("More than 96 tics off: %d",delta);
           error_display = error_display + 1;
        end

      if( shadow_secs == 0 ) begin
         expecting_done_beep = 1;
         delay(1.5);
         verify_cooking;
         if( expecting_done_beep ) begin
            $display("End of cooking beep missing.");
            error_beep = error_beep + 1;
         end
         shadow_state = ss_reset;
         shadow_digits = 0;
      end // if ( shadow_secs == 0 )

   end else if ( shadow_state == ss_pause ) begin

      tosecs(mod_secs);
      if( mod_secs !== shadow_secs ) begin
         $display("Wrong count. (paused)");
         error_display = error_display + 1;
      end

   end else if ( watch_display ) begin

      if( {dmt,dmu,dst,dsu} !== shadow_display
          && {dmt,dmu,dst,dsu} !== alt_display ) begin
        $display("Wrong display, should be %h",shadow_display);
         error_display = error_display + 1;
      end

   end

end

//  Reset shadow state and expected outputs maintained by
//  testbench.
//
task to_reset;
   begin
      shadow_digits = 0;
      shadow_state = ss_reset;
      alt_display = shadow_display;
      shadow_display = 0;
      shadow_power = 10;
      watch_display = 1;
   end
endtask // to_reset

// Send keys to module, update correct state and expected output information.
//
task command;

   input [799:0] cmd;

   integer initialized;
   integer c;
   integer consec_reset;
   reg [5:0] to_key [0:255];
   reg [5:0] key;
```

```
begin

   if( initialized === 'bx ) begin

      for( c = 0; c < 256; c = c + 1 ) to_key[c] = key_never;
      for( c = 0; c < 10;  c = c + 1 ) to_key[ "0" + c ] = key_0 + c;

      to_key["s"] = key_start;
      to_key["r"] = key_reset;
      to_key["p"] = key_power;
      to_key[" "] = key_none;
      to_key[0]   = key_none;

      initialized = 1;
      consec_reset = 0;

   end // if ( initialized === 'bx )

   while( cmd ) begin:COMMAND_LOOP
      reg [7:0] c;
      reg [5:0] key_type;

      c = cmd[799:792];
      key_mod = to_key[ c ];
      key = key_mod;
      key_type = ( c >= "0" && c <= "9" ) ? kty_digit : key_mod;

      if( key == key_never ) begin
         $display("Testbench error: illegal key in command, %s (%d)",c,c);
         $stop;
      end

      casez( {shadow_state,key_type} )

        {3'b???,key_none}:;

        {ss_reset,key_reset}:
          begin
             to_reset;
          end

        {ss_reset,kty_digit}:
          begin
             shadow_digits = 1;
             shadow_state = ss_digit1;
             alt_display = shadow_display;
             shadow_display = key-key_0;
          end

        {ss_pause,key_reset},
        {ss_digit1,key_reset},
        {ss_digit2,key_reset}:
          begin
             to_reset;
          end

        {ss_digit1,kty_digit}:
          begin
             shadow_digits = 2;
             shadow_state = ss_digit2;
             alt_display = shadow_display;
             shadow_display = (shadow_display << 4) | key-key_0;
          end
```

```
    {ss_digit1,key_power}:
      begin
         shadow_state = ss_digit2;
         shadow_digits = 0;
         alt_display = shadow_display; // Power level.
         shadow_power = shadow_display;
         shadow_display = 0;
      end

    {ss_digit1,key_start},{ss_digit2,key_start}:
      begin
         if( shadow_display[7:4] > 5 )
           begin
              expected_beep_done = tics + 16;
           end else begin
              tosecs(shadow_secs);
              start_tics = tics;
              shadow_tics = 0;
              mag_on_total = 0;
              mag_on_start = 0;
              shadow_state = ss_cook;
           end
      end

    {ss_digit2,kty_digit}:
      if( shadow_digits == 4 )
        expected_beep_done = tics + 16;
      else
        begin
           shadow_digits = shadow_digits + 1;
           // If shadow_display zero then power was pressed.
           if( shadow_display ) alt_display = shadow_display;
           shadow_display = (shadow_display << 4) | key-key_0;
        end

    {ss_cook,key_reset}:
      begin
         shadow_state = ss_pause;
         pause_tics = tics;
      end

    {ss_pause,key_start}:
      begin
         verify_cooking;
         mag_on_start = 0;
         shadow_tics = 0;
         mag_on_total = 0;
         start_tics = tics;
         shadow_state = ss_cook;
      end

    default:
      begin
         if( expected_beep_done && expected_beep_done < tics )
           begin
              $display("Missed a beep. (overlap)");
              error_beep = error_beep + 1;
           end
         expected_beep_done = tics + 16;
      end

  endcase // casez( {shadow_state,key_type} )

  @( posedge clk ) @( negedge clk );
  repeat ( $random() & 15 + 3 ) @( negedge clk );
```

```
            key_mod = key_none;
            @( posedge clk ) @( negedge clk );
            @( posedge clk ) @( negedge clk );

            if( key != key_none )
              consec_reset = key == key_reset ? consec_reset + 1 : 0;

            if( monitor_keys && key != key_none && consec_reset < 2 )
              $display("Key %s  State %d, Display: %d%d:%d%d  B %d",
                       c,shadow_state, dmt,dmu,dst,dsu,beep);

            cmd = cmd << 8;

         end // block: COMMAND_LOOP

      end

   endtask // command

   // Reset oven module either using reset line or
   // reset button.
   //
   task reset_oven;
      input hard;
      begin
         if( 0 && hard ) begin
            reset = 1;
            fork:F
              begin repeat ( 256 ) @( clk ); disable F; end
               wait( !beep );
               wait( !mag_on );
            join
            reset = 0;
         end else begin
            command("rrrrrr");
            fork
               wait( !beep );
               wait( !mag_on );
            join
            command("rrrrrr");
         end // else: !if( 0 && hard )
         if( beep || mag_on )
           begin
              $display("Could not reset oven.");
           end
      end
   endtask // reset_oven

   // Actions to be done at the end of a test.
   //
   task endtest;
      input [159:0] name;

      begin

         if( expected_beep_done )
           begin
              if( expected_beep_done >= tics )
                $display("Testbench not waiting long enough for beep.");
              $display("Missed a beep.");
              error_beep = error_beep + 1;
              expected_beep_done = 0;
           end

         if( {dmt,dmu,dst,dsu}!==0 )
```

```
            begin
               $display("Expected to be finished.");
               error_display = error_display + 1;
            end

         watch_display = 0;

         $display("Test %s completed.  (dsp,beep,mag) (%d,%d,%d)",
                  name,
                  error_display, error_beep, error_mag);

         error_total = error_total + error_display + error_beep + error_mag;

         reset_oven(0);

         error_display = 0;
         error_beep = 0;
         error_mag = 0;

      end

   endtask // endtest

   task delay;
      input [63:0] secs;
      #( secs * 1000000 );
   endtask // delay

   initial begin
      monitor_display = 0;
      monitor_keys = 0;
      patient = 0;
      expected_beep_done = 0;
      expecting_done_beep = 0;
      error_display = 0;
      error_beep = 0;
      error_mag = 0;
      error_total = 0;

      #1;
      to_reset;
      reset_oven(1);

      command("50prr");
      delay(40);
      endtest("Power Too High");

      command("12s"); delay(16);
      endtest("Basic");

      if( patient ) begin
         $display("Starting test Long, be patient or modify testbench.");
         command("100s"); delay(90*60+5);
         endtest("Long");
      end

      command("30s"); delay(14);
      command("1"); delay(2); command("p");
      delay(20);
      endtest("Basic Disturbed");

      command("5p30s");
      delay(40);
      endtest("Half Power");
```

```
        command("9p3p0s");
        delay(40);
        endtest("Power Twice");

        command("3ppp19s");
        delay(40);
        endtest("Power Thrice");

        command("20s");
        delay(10);
        command("r");
        delay(5);
        command("s");
        delay(10);
        endtest("Reset Start");

        command("20s");
        delay(10);
        command("r");
        delay(5);
        command("r");
        delay(1);
        endtest("Reset Reset");

        command("s");
        delay(5);
        endtest("Null Start");

        command("ps");
        delay(5);
        endtest("Null Power Start");

        command("7p30s"); delay(10);
        command("1"); delay(2); command("s"); delay(3);
        command("12344321");
        delay(40);
        endtest("Power Disturbed");

        command("12r5s"); delay(10);
        endtest("Twelve no 5");

        command("90s"); delay(1); command("rr");
        endtest("Ninety Seconds");

        command("12345rr");
        endtest("Display Overflow");

        $display("All tests completed, %d total errors.",error_total);

        $stop;
    end // initial begin

endmodule // test_oven
```

```
////////////////////////////////////////////////////////////////////////////////
///
/// Solution to LSU EE 4702-1 Spring 2001 Homework 4
///
///
//  Includes a testbench (which was not graded).


`timescale 1us/1us

module microwave_oven_controller(beep,dmt,dmu,dst,dsu,mag_on,
                                 key_code,reset,clk);
   input key_code;              // Key begin pressed (see parameters).
                                // Can be tested on the positive edge of clk.
   input reset;                 // Reset signal. Can be tested on posedge clk.
   input clk;                   // A 64 Hz clock. (Did Edison consider it?)
   output mag_on;               // When 1, magnetron is on (oven is heating).
   output beep;                 // When 1, emit tone.
   output dmt;                  // Tens digit of minute display.
   output dmu;                  // Units digit of minute display.
   output dst, dsu;             // Tens and units digits of seconds display.

   wire   clk;
   wire [5:0] key_code;
   reg [3:0]  dmt, dmu, dst, dsu;
   reg        mag_on;
   reg        beep;

   parameter  key_none  = 6'd0;  // No key pressed.
   parameter  key_never = 6'd1;  // This code will never be returned.
   parameter  key_start = 6'd10;
   parameter  key_reset = 6'd11;
   parameter  key_power = 6'd12;

   parameter  key_0 = 6'd20;
   parameter  key_1 = 6'd21;
   parameter  key_2 = 6'd22;
   parameter  key_3 = 6'd23;
   parameter  key_4 = 6'd24;
   parameter  key_5 = 6'd25;
   parameter  key_6 = 6'd26;
   parameter  key_7 = 6'd27;
   parameter  key_8 = 6'd28;
   parameter  key_9 = 6'd29;

   parameter  kty_digit = 6'd30;

   parameter  st_reset    = 0;
   parameter  st_entry    = 1;
   parameter  st_entry_p1 = 2;
   parameter  st_entry_p2 = 5;
   parameter  st_heating  = 3;
   parameter  st_paused   = 4;

   reg [2:0]  digit_count;
   reg [3:0]  power;
   reg [5:0]  key_type, last_key;
   reg [3:0]  state, next_state;
   reg [7:0]  beep_timer;

   task add_digit;
      begin
         if( digit_count == 4 )
```

```
          beep_timer = 16;
        else begin
           dmt = dmu;
           dmu = dst;
           dst = dsu;
           dsu = key_code - key_0;
           digit_count = digit_count + 1;
        end
     end
  endtask // add_digit

  task do_reset;
     begin
        dmt = 0;
        dmu = 0;
        dst = 0;
        dsu = 0;
        digit_count = 0;
        beep_timer = 0;
        next_state = st_reset;
        mag_on = 0;
     end
  endtask // do_reset

  reg [7:0]  on_timer, off_timer;
  reg [5:0]  sec_timer;

  always @( posedge clk )
    if( reset ) begin

       do_reset;
       state     = st_reset;
       sec_timer = 0;
       on_timer  = 0;
       off_timer = 0;
       last_key  = key_none;
       beep      = 0;

    end else begin // if ( reset )

       if( key_code != key_none && last_key == key_none ) begin

          if( key_code >= key_0 && key_code <= key_9 )
            key_type = kty_digit;
          else
            key_type = key_code;

          casez( {state, key_type} )

            {st_reset,kty_digit}:
              begin
                 add_digit;
                 power = 10;
                 next_state = st_entry;
              end

            {st_entry_p1,kty_digit}:
              begin
                 dsu = 0;
                 add_digit;
                 next_state = st_entry_p2;
              end

            {st_entry_p2,kty_digit}:
              begin
```

```
            add_digit;
            next_state = state;
          end

        {st_entry,kty_digit}:
          begin
            add_digit;
            next_state = state;
          end

        {st_entry,key_power}:
          begin
            if( digit_count == 1 )
              power = dsu;
            else
              beep_timer = 16;
            next_state = st_entry_p1;
          end

        {st_entry_p2,key_start}, {st_entry,key_start}:
          begin
            next_state = st_heating;
          end


        {st_paused,key_start}:
          begin
            next_state = st_heating;
          end

        // Leonardo incorrectly infers parallel case, so need
        // to test state.
        {4'b????,key_reset}:
          if( state == st_heating )
            next_state = st_paused;
          else
            do_reset;

        default:
          begin
            next_state = state; beep_timer = 16;
          end

      endcase // casez( {state, key_type} )

    end // if ( key_code != key_none && last_key == key_none )

    // next_state may be reassigned below.

    sec_timer = sec_timer + 1;

    beep = | beep_timer;
    if( beep_timer ) beep_timer = beep_timer - 1;

    if( state == st_heating ) begin

      if( {dmt,dmu,dst,dsu} == 0 ) begin
        beep_timer = 128;
        mag_on = 0;
        next_state = st_reset;
      end else begin
        if( !sec_timer ) begin:T
          if( dsu ) begin dsu = dsu - 1; disable T; end
          dsu = 9;
          if( dst ) begin dst = dst - 1; disable T; end
```

```
                dst = 5;
                if( dmu ) begin dmu = dmu - 1; disable T; end
                dmu = 9;
                if( dmt ) dmt = dmt - 1;
             end

             if( on_timer )       on_timer  = on_timer - 1;
             else if( off_timer ) off_timer = off_timer - 1;

             mag_on = |on_timer | ~|off_timer;

             if( !on_timer && !off_timer ) begin
                on_timer  = power * 16;
                off_timer = 160 - on_timer;
             end

          end // else: !if( {dmt,dmu,dst,dsu} == 0 )
       end else begin // if ( state == st_heating )
          on_timer = 0;
          off_timer = 0;
          mag_on = 0;
       end // else: !if( state == st_heating )

       last_key = key_code;
       state = next_state;

    end // else: !if( reset )

endmodule // microwave_oven_controller


// exemplar translate_off


module test_oven();

   reg  clk;
   wire [3:0] dmt, dmu, dst, dsu;
   wire       mag_on;
   reg [5:0]  key_mod;
   reg        reset;

   // Set this to 1 to have each change in the oven display appear
   // on the console.
   reg        monitor_display;
   // Set this to one to have each key press appear on the console.
   reg        monitor_keys;
   reg        monitor_beep;
   reg        monitor_mag;
   // Set this to one to get long test.
   reg        patient;

   microwave_oven_controller oven(beep,dmt,dmu,dst,dsu,mag_on,key_mod,reset,clk);
   time       tics;

   wire [15:0] mod_digits = {dmt,dmu,dst,dsu};

   initial tics = 0;
   always begin clk = 0; #5625; tics = tics + 1; #0; clk=1; #10000; end

   parameter   ss_reset  = 3'd0;
   parameter   ss_digit1 = 3'd1; // Single digit, power not entered.
   parameter   ss_digit2 = 3'd2; // Power entered or > 1 digit.
   parameter   ss_cook   = 3'd3;
   parameter   ss_pause  = 3'd4;
```

```
   parameter   kty_digit = 6'd30;

   reg [15:0]  shadow_display, alt_display;
   integer     alt_disp_stale;
   reg [2:0] shadow_state;
   integer   shadow_secs, mod_secs, delta;
   integer   shadow_tics, pause_tics, start_tics;
   integer   shadow_power, shadow_digits;
   integer   expected_beep_done, expecting_done_beep;

   integer   watch_display;

   reg [7:0] error_display, error_mag, error_beep;
   integer   error_total;
   reg [7:0]   error_beep_total, error_mag_total, error_display_total;

   parameter key_none  = 6'd0;  // No key pressed.
   parameter  key_never = 6'd1;  // This code will never be returned.
   parameter  key_start = 6'd10;
   parameter  key_reset = 6'd11;
   parameter  key_power = 6'd12;

   parameter  key_0 = 6'd20;
   parameter  key_1 = 6'd21;
   parameter  key_2 = 6'd22;
   parameter  key_3 = 6'd23;
   parameter  key_4 = 6'd24;
   parameter  key_5 = 6'd25;
   parameter  key_6 = 6'd26;
   parameter  key_7 = 6'd27;
   parameter  key_8 = 6'd28;
   parameter  key_9 = 6'd29;

   parameter  show_key = 0;

   function integer abs;
      input a;
      integer a;
      abs = a < 0 ? -a : a;
   endfunction // abs


`define check_digit(d,l)                                                \
   if( (d) > (l) || (d) < 0 ) begin                                     \
      error_display = error_display | 1;                                \
      secs = -1;                                                        \
      disable tosecs;                                                   \
   end

   // Convert time on display to seconds.
   //
   task tosecs;
      output secs;
      integer secs;

      begin
         `check_digit(dmt,9)
         `check_digit(dmu,9)
         `check_digit(dst,5)
         `check_digit(dsu,9)
         secs = dmt * 600 + dmu * 60 + dst * 10 + dsu;
      end

   endtask // tosecs
```

```
/// Listen Beep
//
always @( beep )
  if( beep ) begin
     if( monitor_beep ) $display("Beep starting.");
     if( expecting_done_beep )
       begin
          expected_beep_done = 128;
          expecting_done_beep = 0;
       end
     if( expected_beep_done == 0 ) begin
        $display("Should not be beeping.");
        error_beep = error_beep | 1;
     end
  end else begin:B // if ( beep )
     integer delta;
     if( monitor_beep ) $display("Beep ending.");
     delta = abs( tics - expected_beep_done );
     if( shadow_state != ss_reset
         && expected_beep_done && delta > 5 ) begin
        $display("Beep wrong time. %d",delta);
        error_beep = error_beep | 2;
     end
     expected_beep_done = 0;
  end

/// Watch Magnetron
//
integer mag_on_start, mag_on_total;

always @( mag_on )
  if( mag_on ) begin
     if( monitor_mag ) $display("Mag on.");
     if( shadow_state != ss_cook )
       begin
          $display("Mag on when cooking off.");
          error_mag = error_mag | 1;
       end
     if( mag_on_start != 0 ) begin:A
        integer cycle_this;
        cycle_this = tics - mag_on_start;
        if( shadow_power > 0 && shadow_power < 10
            && abs( cycle_this - 160 ) > 10 )
          begin
             $display("Mag cycle error.");
             error_mag = error_mag | 2;
          end
     end
     mag_on_start = tics;
  end else begin // if ( mag_on )
     if( monitor_mag ) $display("Mag off.");
     mag_on_total = mag_on_total + tics - mag_on_start;
  end // else: !if( mag_on )

/// Verify correct magnetron-on time.
//
task verify_cooking;
   begin:A
      integer correct_mag_tics;
      integer delta;

      correct_mag_tics = shadow_tics * shadow_power / 10;
```

```
        delta = abs(correct_mag_tics - mag_on_total);

        if( mag_on ) begin
           $display("Mag should be off.");
           error_mag = error_mag | 'h10;
        end

        if( delta > 128 ) begin
           $display("Wrong power level. %d %d ",
                    correct_mag_tics, mag_on_total);
           error_mag = error_mag | 4;
        end
     end // block: A
  endtask // verify_cooking

  ///  Watch display, etc.
  //
  always @( dmt or dmu or dst or dsu or shadow_display ) #1 begin

     if( monitor_display )
       $display("Display: %h   sh: %h  alt: %h   secs %d, state %d",
                mod_digits, shadow_display,alt_display,
                shadow_secs, shadow_state);

     if( shadow_state == ss_cook ) begin

        shadow_secs = shadow_secs - 1;
        shadow_tics = shadow_tics + 64;

        tosecs(mod_secs);

        if( mod_secs !== shadow_secs )
          begin
             $display("Wrong count. (cooking)");
             error_display = error_display | 4;
          end

        delta = shadow_tics - ( tics - start_tics );
        if( abs(delta) > 96 )
          begin
             $display("More than 96 tics off: %d",delta);
             error_display = error_display | 2;
          end

        if( mod_digits == 0 ) begin
           expecting_done_beep = 1;
           delay(1.5);
           verify_cooking;
           if( expecting_done_beep ) begin
              $display("End of cooking beep missing.");
              error_beep = error_beep | 4;
           end
           shadow_state = ss_reset;
           shadow_display = 0;
        end else if ( shadow_secs == 0 )
          begin
             $display("Count problem.");
          end

     end else if ( shadow_state == ss_pause ) begin

        tosecs(mod_secs);
        if( mod_secs !== shadow_secs ) begin
           $display("Wrong count. (paused)");
           error_display = error_display | 8;
```

```
      end

   end else if ( watch_display ) begin

      if( mod_digits !== shadow_display
          && ( alt_disp_stale <= tics || mod_digits !== alt_display ) )
        begin
         $display("Wrong display, should be %h or maybe %h but not %h",
                  shadow_display,alt_display,mod_digits);
         error_display = error_display | 'h10;
      end

   end

end

//  Reset shadow state and expected outputs maintained by
//  testbench.
//
task to_reset;
   begin
      shadow_digits = 0;
      alt_display = shadow_display;
      alt_disp_stale = tics + 2;
      shadow_state = ss_reset;
      shadow_display = 0;
      shadow_power = 10;
      watch_display = 1;
   end
endtask // to_reset

// Send keys to module, update correct state and expected output information.
//
task command;

   input [799:0] cmd;

   integer initialized;
   integer c;
   integer consec_reset;
   reg [5:0] to_key [0:255];
   reg [5:0] key;

   begin

      if( initialized === 'bx ) begin

         for( c = 0; c < 256; c = c + 1 ) to_key[c] = key_never;
         for( c = 0; c < 10;  c = c + 1 ) to_key[ "0" + c ] = key_0 + c;

         to_key["s"] = key_start;
         to_key["r"] = key_reset;
         to_key["p"] = key_power;
         to_key[" "] = key_none;
         to_key[0]   = key_none;

         initialized = 1;
         consec_reset = 0;

      end // if ( initialized === 'bx )

      while( cmd ) begin:COMMAND_LOOP
         reg [7:0] c;
         reg [5:0] key_type;
```

```
c = cmd[799:792];
cmd = cmd << 8;

if( c == 0 || c == " " ) disable COMMAND_LOOP;

key_mod = to_key[ c ];
key = key_mod;
key_type = ( c >= "0" && c <= "9" ) ? kty_digit : key_mod;

if( key == key_never ) begin
   $display("Testbench error: illegal key in command, %s (%d)",c,c);
   $stop;
end

if( key != key_none )
  consec_reset = key == key_reset ? consec_reset + 1 : 0;

if( monitor_keys && key != key_none && consec_reset < 3 )
  $display("Key %s  ", c);

casez( {shadow_state,key_type} )

  {3'b???,key_none}:;

  {ss_reset,key_reset}:
    begin
       to_reset;
    end

  {ss_reset,kty_digit}:
    begin
       shadow_digits = 1;
       shadow_state = ss_digit1;
       alt_display = shadow_display;
       alt_disp_stale = tics + 2;
       shadow_display = key-key_0;
    end

  {ss_pause,key_reset},
  {ss_digit1,key_reset},
  {ss_digit2,key_reset}:
    begin
       to_reset;
    end

  {ss_digit1,kty_digit}:
    begin
       shadow_digits = 2;
       shadow_state = ss_digit2;
       alt_display = shadow_display;
       alt_disp_stale = tics + 2;
       shadow_display = (shadow_display << 4) | key-key_0;
    end

  {ss_digit1,key_power}:
    begin
       shadow_state = ss_digit2;
       shadow_digits = 0;
       alt_display = shadow_display; // Power level.
       alt_disp_stale = 'h7fffffff;
       shadow_power = shadow_display;
       shadow_display = 0;
    end

  {ss_digit1,key_start},{ss_digit2,key_start}:
```

```
                begin
                   tosecs(shadow_secs);
                   start_tics = tics;
                   shadow_tics = 0;
                   mag_on_total = 0;
                   mag_on_start = 0;
                   shadow_state = ss_cook;
                end

              {ss_digit2,kty_digit}:
                if( shadow_digits == 4 )
                  expected_beep_done = tics + 16;
                else
                  begin
                     shadow_digits = shadow_digits + 1;
                     // If shadow_display zero then power was pressed.
                     alt_disp_stale = tics + 2;
                     alt_display = mod_digits;
                     shadow_display = (shadow_display << 4) | key-key_0;
                  end

              {ss_cook,key_reset}:
                begin
                   shadow_state = ss_pause;
                   shadow_display = mod_digits;
                   pause_tics = tics;
                end

              {ss_pause,key_start}:
                begin
                   verify_cooking;
                   mag_on_start = 0;
                   shadow_tics = 0;
                   mag_on_total = 0;
                   start_tics = tics;
                   shadow_state = ss_cook;
                end

              default:
                begin
                   if( expected_beep_done && expected_beep_done < tics )
                     begin
                        $display("Missed a beep. (overlap)");
                        error_beep = error_beep | 8;
                     end
                   expected_beep_done = tics + 16;
                end

            endcase // casez( {shadow_state,key_type} )

            @( posedge clk ) @( negedge clk );
            repeat ( $random() & 15 + 3 ) @( negedge clk );
            key_mod = key_none;
            @( posedge clk ) @( negedge clk );
            @( posedge clk ) @( negedge clk );

         end // block: COMMAND_LOOP

      end

   endtask // command

   // Reset oven module either using reset line or
   // reset button.
   //
```

```
task reset_oven;
   input hard;
   begin
      if( hard ) begin
         reset = 1;
         fork:F
           begin repeat ( 5 * 60 * 64 ) @( posedge clk ); disable F; end
            wait( !beep );
            wait( !mag_on );
         join
         reset = 0;
      end else begin
         command("rrrrrr");
         fork:T
            wait( !beep );
            wait( !mag_on );
            begin repeat ( 5 * 60 * 64 ) @( posedge clk ); disable T; end
         join
         command("rrrrrr");
      end // else: !if( 0 && hard )
      if( beep || mag_on )
        begin
           $display("Could not reset oven.");
           if( beep ) error_beep = error_beep | 'h20;
           if( mag_on ) error_mag = error_mag | 8;
        end
   end
endtask // reset_oven

// Actions to be done at the end of a test.
//
task endtest;
   input [159:0] name;
   integer error_count;

   begin

      if( expected_beep_done )
        begin
           if( expected_beep_done >= tics )
             $display("Testbench not waiting long enough for beep.");
           $display("Missed a beep.");
           error_beep = error_beep | 'h10;
           expected_beep_done = 0;
        end

      if( mod_digits!==0 )
        begin
           $display("Display should be zero.");
           error_display = error_display + 'h20;
        end

      verify_cooking;

      watch_display = 0;

      error_count = ( error_display ? 1 : 0 )
      + ( error_beep ? 1 : 0 ) + ( error_mag ? 1 : 0 );
      error_total = error_total + error_count;

      error_beep_total = error_beep_total | error_beep;
      error_mag_total = error_mag_total | error_mag;
      error_display_total = error_display_total | error_display;

      $display("OUTCOME: %s on test %s.  (dsp,beep,mag) (%2h,%2h,%2h)",
```

```
                error_count ? "FAIL" : "PASS",
                name,
                error_display, error_beep, error_mag);

        reset_oven(0);

        error_display = 0;
        error_beep = 0;
        error_mag = 0;
        mag_on_total = 0;
        shadow_tics = 0;

    end

endtask // endtest


task delay;
    input [63:0] secs;
    #( secs * 1000000 );
endtask // delay

initial begin
    monitor_display = 1;
    monitor_keys = 1;
    monitor_beep = 1;
    monitor_mag = 1;
    patient = 1;
    expected_beep_done = 0;
    expecting_done_beep = 0;
    error_display = 0;
    error_display_total = 0;
    error_beep = 0;
    error_beep_total = 0;
    error_mag = 0;
    error_mag_total = 0;
    error_total = 0;
    mag_on_total = 0;

    #1;
    to_reset;
    reset_oven(1);

    error_display = 0;
    error_display_total = 0;
    error_beep = 0;
    error_beep_total = 0;
    error_mag = 0;
    error_mag_total = 0;
    error_total = 0;
    mag_on_total = 0;

    command("50prr");
    delay(54);
    endtest("Power Too High");

    command("32s"); delay(36);
    endtest("Basic");

    if( patient ) begin:PATIENT
        integer old_mon, old_mag;
        old_mon = monitor_display;
        old_mag = monitor_mag;
        monitor_display = 0;
        $display("Starting test Long, be patient or modify testbench.");
```

```
      command("1234s"); delay(14*60);
      endtest("Long");
      monitor_display = old_mon;
      monitor_mag = old_mag;
    end

    command("30s"); delay(14);
    command("1"); delay(2); command("p");
    delay(20);
    endtest("Basic Disturbed");

    command("5p30s");
    delay(40);
    endtest("Half Power");

    command("9p3p0s");
    delay(40);
    endtest("Power Twice");

    command("3ppp19s");
    delay(40);
    endtest("Power Thrice");

    command("20s");
    delay(10);
    command("r");
    delay(5);
    command("s");
    delay(16);
    endtest("Reset Start");

    command("20s");
    delay(10);
    command("r");
    delay(5);
    command("r");
    delay(16);
    endtest("Reset Reset");

    command("s");
    delay(5);
    endtest("Null Start");

    command("ps");
    delay(5);
    endtest("Null Power Start");

    command("7p30s"); delay(10);
    command("1"); delay(2); command("s"); delay(3);
    command("12344321");
    delay(40);
    endtest("Power Disturbed");

    command("12r5s"); delay(135);
    endtest("Twelve no 5");

    command("12345rr"); delay(5);
    endtest("Display Overflow");

    if( show_key ) begin

    $display("\n ** Error Codes **
Display (Codes in hexadecimal. Error codes or'ed together.)
  1  Digit out of range. (Check digit.)
  2  Display digit change more than 96 tics off.
```

```
    4  Wrong count. (cooking)
    8  Wrong count. (paused)
   10  Wrong count displayed. (Not cooking nor paused.)
   20  Display should be zero.

  Beep:
    1  Should not be beeping.
    2  Beep duration wrong.
    4  End of cooking beep missing.
    8  Missed a beep. (overlap)
   10  Missed a beep. (endtest)
   20  Wouldn't stop beeping!

  Mag
    1  Mag on when cooking off.
    2  Mag cycle error.
    4  Wrong power level.
    8  Would not turn off.
   10  Should be off (when cooking verified).

");

      $display("\n ** Testbench Monitoring ** ");
      $display("    Display %s", monitor_display ? "on" : "off");
      $display("    Keys %s", monitor_keys ? "on" : "off");
      $display("    Beep %s", monitor_beep ? "on" : "off");
      $display("    Mag %s", monitor_mag ? "on" : "off");
      $display("    To turn monitoring on and off edit monitor_FOO variables.");
      $display("    at last \"initial begin\" in testbench.\n\n");
      end

      $display("%s, beep tests (code: %h).",
               error_beep_total ? "FAIL" : "PASS", error_beep_total);
      $display("%s, mag tests (code: %h).",
               error_mag_total ? "FAIL" : "PASS", error_mag_total);
      $display("%s, display tests (code: %h).",
               error_display_total ? "FAIL" : "PASS", error_display_total);

      $display("All tests completed, %d total errors.",error_total);

      $stop;
   end // initial begin

endmodule // test_oven

// exemplar translate_on
```

```
////////////////////////////////////////////////////////////////////////////////
///
/// Solution to LSU EE 4702-1 Spring 2001 Homework 5
///

  // Assignment: http://www.ee.lsu.edu/v/2001/hw05.pdf

////////////////////////////////////////////////////////////////////////////////


// The log_2 of the number of numbers stored by the bsearch module.
// Keep this small to limit synthesis time.
`define SIZELG 4
`define SIZE (1<<`SIZELG)       // The number of numbers stored by bsearch.
`define SIZERANGE `SIZELG-1:0
`define ELEMBITS 8              // The number of bits in each number stored.
`define ELEMRANGE `ELEMBITS-1:0
`define MAXELEM (1<<`ELEMBITS)


////////////////////////////////////////////////////////////////////////////////
//
// Original bsearch module.
//
// Not part of the solution, here for comparison to the other modules.
// This module is synthesizable (despite misleading macro names).  The
// synthesized hardware does the entire lookup in one cycle, which
// requires alot of hardware.

// Number of gates :                    5365
// clk                     : 50.7 MHz
// Critical path: num_2_ -> result[1], 19.18 ns

`define xNOT_SYN
`ifdef NOT_SYN
module bsearch(result,din,op,reset,clk);
   input din, op, reset, clk;
   output result;
   wire [7:0] din;
   wire [2:0] op;
   reg [2:0]  result;

  `include "bsearch_names.v"

   reg [7:0]  dtable [0:`SIZE-1];
   reg [`SIZELG:0] num;
   reg [`SIZERANGE] current, try, delta;
   reg [`ELEMRANGE] trydata;
   reg              match;

   // Bug workaround.  Needed to avoid a synthesis bug.
   wire [`ELEMRANGE] bug_workaround = dtable[1];

   always @( posedge clk )

     if( reset ) begin

        num = 0;
        result = re_r_idle;

     end else begin

        case( op )
```

```
            op_nop:;

            op_reset:
              begin
                 num = 0;
                 result = re_r_idle;
              end

            op_insert:
              if( num == `SIZE ) begin
                 result = re_i_full;
              end else if( num > 0 && dtable[num-1] >= din ) begin
                 result = re_i_misordered;
              end else begin
                 dtable[num] = din;
                 num = num + 1;
                 result = re_i_inserted;
              end

            op_find:
              begin
                 match   = 0;
                 current = 0;
                 delta   = 1 << ( `SIZELG - 1 );
                 begin:BLOOP forever begin
                    try = current | delta;
                    if( try < num ) begin
                       trydata = dtable[try];
                       match = trydata == din;
                       if( match ) disable BLOOP;
                       if( trydata < din ) current = try;
                    end
                    if( !delta ) disable BLOOP;
                    delta = delta >> 1;
                 end end

                 result = match ? re_f_present : re_f_absent;

              end

          endcase

       end

endmodule
`endif

////////////////////////////////////////////////////////////////////////////////
//
// Problem 1:  Form 2 bsearch Module
//
// Synthesizable Form 2.
//
//
// Clock Frequency (MHz):                113.5 MHz
// Area (number of gates):              4275
// Worst-case time to find a number:    6 / 113.5 MHz = 52.9 ns
//

`define xFORM2
`ifdef FORM2

module bsearch(result,din,op,reset,clk);
   input din, op, reset, clk;
```

```
  output result;
  wire [7:0] din;
  wire [2:0] op;
  reg [2:0]  result;

`include "bsearch_names.v"

  reg        looping;

  reg [7:0]  dtable [0:`SIZE-1];
  reg [`SIZELG:0] num;
  reg [`SIZERANGE] current, try, delta;
  reg [`ELEMRANGE] trydata;
  reg              match;

  wire [`ELEMRANGE] bug_workaround = dtable[12];

  always @( posedge clk )

    if( reset ) begin

       looping = 0;
       num     = 0;
       result  = re_r_idle;

    end else if( looping ) begin

       try = current + delta;
       if( try < num ) begin
          trydata = dtable[try];
          match = trydata == din;
          if( match ) looping = 0;
          if( trydata < din ) current = try;
       end
       if( !delta ) looping = 0;
       delta = delta >> 1;
       if( !looping ) result = match ? re_f_present : re_f_absent;

    end else begin

       case( op )

         op_nop:;

         op_reset:
           begin
              num    = 0;
              result = re_r_idle;
           end

         op_insert:
           if( num == `SIZE ) begin
              result = re_i_full;
           end else if( num > 0 && dtable[num-1] >= din ) begin
              result = re_i_misordered;
           end else begin
              dtable[num] = din;
              num = num + 1;
              result = re_i_inserted;
           end

         op_find:
           begin
              match   = 0;
              current = 0;
```

```
                delta   = 1 << ( `SIZELG - 1 );
                result  = re_busy;
                looping = 1;
              end

          endcase

      end

endmodule
`endif


////////////////////////////////////////////////////////////////////////////
//
// Problem 2:  Form 3 bsearch Module
//
// Synthesizable  Form 3.
// Module does one dtable lookup per cycle.
//
// Clock Frequency (MHz):                  109.1 MHz
// Area (number of gates):                4155
// Worst-case time to find a number:  5 / 109.1 MHz = 45.8 ns
//
// Critical path: reg_delta(1)/XQ to result[1], 8.47 ns


`define xFORM3
`ifdef FORM3
module bsearch(result,din,op,reset,clk);
   input din, op, reset, clk;
   output result;
   wire [7:0] din;
   wire [2:0] op;
   reg [2:0]  result;

 `include "bsearch_names.v"

   reg [7:0]  dtable [0:`SIZE-1];
   reg [`SIZELG:0] num;
   reg [`SIZERANGE] current, try, delta;
   reg [`ELEMRANGE] trydata;
   reg              match;

   always @( posedge clk )

     if( reset ) begin

        num = 0;
        result = re_r_idle;

     end else begin

        case( op )

          op_nop:;

          op_reset:
            begin
               num = 0;
               result = re_r_idle;
            end

          op_insert:
            if( num == `SIZE ) begin
```

```
               result = re_i_full;
            end else if( num > 0 && dtable[num-1] >= din ) begin
               result = re_i_misordered;
            end else begin
               dtable[num] = din;
               num = num + 1;
               result = re_i_inserted;
            end

         op_find:
            begin
               match   = 0;
               current = 0;
               delta   = 1 << ( `SIZELG - 1 );
               result  = re_busy;

               begin:BLOOP forever begin
                  @( posedge clk );
                  try = current + delta;
                  if( try < num ) begin
                     trydata = dtable[try];
                     // This would split the delta -> result path between two
                     // cycles, but it would take nearly twice as long to find
                     // a result.
                     //  @( posedge clk );
                     match = trydata == din;
                     if( match ) disable BLOOP;
                     if( trydata < din ) current = try;
                  end
                  if( !delta ) disable BLOOP;
                  delta = delta >> 1;
               end end
               result = match ? re_f_present : re_f_absent;

            end

      endcase

   end

endmodule
`endif



////////////////////////////////////////////////////////////////////////////////
//
// Problem 3:  Form 3 bsearch Module, Speed Enhanced
//
// Synthesizable, Form 3.
// Module does two dtable lookups per cycle.
//

// Several solutions within module, area, time, and performance in comments
// next to code.

// Clock Frequency (MHz):
// Area (number of gates):
// Worst-case time to find a number:
//


`define FORM3_FAST
`ifdef FORM3_FAST
module bsearch(result,din,op,reset,clk);
```

```
  input din, op, reset, clk;
  output result;
  wire [7:0] din;
  wire [2:0] op;
  reg [2:0]  result;

`include "bsearch_names.v"

  reg [7:0]  dtable [0:`SIZE-1];
  reg [`SIZELG:0] num;
  reg [`SIZERANGE] current, try, delta;
  reg [`SIZERANGE] try0, try1, delta2;
  reg [`ELEMRANGE] trydata, trydata0, trydata1;
  reg              match, match0, match1, m;

  always @( posedge clk )

    if( reset ) begin

       num = 0;
       result = re_r_idle;

    end else begin

       case( op )

         op_nop:;

         op_reset:
           begin
              num = 0;
              result = re_r_idle;
           end

         op_insert:
           if( num == `SIZE ) begin
              result = re_i_full;
           end else if( num > 0 && dtable[num-1] >= din ) begin
              result = re_i_misordered;
           end else begin
              dtable[num] = din;
              num = num + 1;
              result = re_i_inserted;
           end

         op_find:
           begin
              current = 0;
              delta   = 1 << ( `SIZELG - 1 );
              result  = re_busy;

`define ALWAYS
 `ifdef ALWAYS
              // Two iterations per clock cycle, dtable lookups done simultaneously.
              // Code below clk                    : 3 / 97.8 MHz = 30.7 ns
               //  Number of gates :                    7513
              delta2 = delta >> 1;
              begin:BLOOP forever begin
                 @( posedge clk );
                 try = current + delta;

                 try0 = current + delta2;
                 try1 = try + delta2;
                 trydata = dtable[try];
                 trydata0 = dtable[try0];
```

```
                trydata1 = dtable[try1];
                match = trydata == din;
                match0 = trydata0 == din;
                match1 = trydata1 == din;
                if( try < num ) begin
                   if( match ) begin m = 1; disable BLOOP; end
                   if( trydata > din )
                     begin
                        if( match0 ) begin m = 1; disable BLOOP; end
                        if( trydata0 < din ) current = try0;
                     end
                   else if( try1 < num )
                     begin
                        if( match1 ) begin m = 1; disable BLOOP; end
                        if( trydata1 < din ) current = try1;
                        else current = try;
                     end
                        else
                          current = try;
                end else if( try0 < num ) begin
                   begin
                      if( match0 ) begin m = 1; disable BLOOP; end
                      if( trydata0 < din ) current = try0;
                   end
                end
                if( !delta ) begin m = 0; disable BLOOP; end
                delta = delta >> 2;
                delta2 = delta2 >> 2;
             end end
`endif

`ifdef NEVER
             // Completely unrolled, all five iterations in one cycle.
             // Code below clk                    : 1/ 55.5 MHz  = 18.0 ns
             //  Number of gates :                    5482
             match = 0;
             repeat ( `SIZELG + 1 ) begin
                try = current + delta;
                if( !match && try < num ) begin
                   trydata = dtable[try];
                   match = trydata == din;
                   if( trydata < din ) current = try;
                end
                delta = delta >> 1;
             end
             m = match;
`endif
`ifdef NEVER
             // Original Code
             //  Code below: clk                   : 5/ 109.1 MHz = 46.2 ns
              //  Number of gates :                   4155
             match = 0;
             begin:BLOOP forever begin
                @( posedge clk );
                try = current + delta;
                if( try < num ) begin
                   trydata = dtable[try];
                   match = trydata == din;
                   if( match ) disable BLOOP;
                   if( trydata < din ) current = try;
                end
                if( !delta ) disable BLOOP;
                delta = delta >> 1;
             end end
             m = match;
```

```
`endif

`ifdef NEVER
                // Two iterations per cycle.
                // Iterations are sequential. (Second dtable lookup after first.)
                 //  clk                    : 3 / 63.7 MHz = 47.1 ns
                 // Number of gates :                    6465
                match = 0;
                begin:BLOOP forever begin
                   @( posedge clk );
                   repeat( 2 ) begin
                      try = current + delta;
                      if( !match && try < num ) begin
                         trydata = dtable[try];
                         match = trydata == din;
                         if( trydata < din ) current = try;
                      end
                      delta2 = delta;  delta = delta >> 1;
                   end
                   if( match ) disable BLOOP;
                   if( !delta2 ) disable BLOOP;
                end end
                m = match;
`endif

                result = m ? re_f_present : re_f_absent;

            end

        endcase

    end

endmodule
`endif


////////////////////////////////////////////////////////////////////////////////
//
// Testbench
//

// Testbench module is named testbsearch.

// The testbench can be copied into this file or another one and
// modified.  The testbench might be updated before the homework
// is due though.

`include "/home/classes/ee4702/files/v/hw05tb.v"
```

26 Spring 2000 Solutions

```
// Solution to LSU EE 4701 Spring 2000 HW 5.


`ifdef ALU

module alu(res,err,a,b,op);
   input a, b, op;
   output res, err;

   parameter op_add = 0, // Addition.
             op_sub = 1, // Subtraction
             op_and = 2; // Bitwise and.

   wire [7:0] a, b;
   wire [2:0]  op;

   reg [7:0]  res;
   reg         err;

   // exemplar full_case
   always @( a or b or op )
     case( op )
       op_add: {err,res} = a + b;
       op_sub: {err,res} = a - b;
       op_and: begin err = 0; res = a & b; end
     endcase // case( op )

endmodule // alu

`endif

module latch_thing(w,x,y,z,a,b,c,d,r,clk);
   input a, b, c, d, r, clk;
   output w, x, y, z;

   reg w, x, y, z;
   wire a, b, c, d, r, clk;

   always @( negedge clk or posedge r ) if( r ) w = 0; else w = d;

   always @( posedge clk or posedge r ) if( r ) y = 0; else y = a;

   always @( clk or c or r or d or b )
     if( r ) z = 0; else if( clk && d == b ) z = c;

   wire deqb = d == a && clk;

   always @( posedge clk or posedge r or posedge deqb )
     if( r )
       x = 0;
     else if( deqb )
       x = 1;
     else if( a ) x = b;

endmodule // latch_thing
```

```
//
// LSU EE 4702-1 Spring 2000 Homework 6 Solution
//


module width_change(out,full,complete,empty,outclk,in,inclk,reset);
   input outclk, in, inclk, reset;
   output out, full, complete, empty;

   parameter storage = 32;

   wire [7:0] out;
   wire [3:0] in;
   wire       inclk, outclk, full, empty, complete;

   reg [storage-1:0] sto;
   reg [1:0]  head_word;
   reg [2:0]  tail_nibble;

   reg        empty_in, empty_out, full_in, full_out;

   assign out      = empty ? 0 : sto >> { head_word, 3'b0 };
   assign empty    = empty_in ^ empty_out;
   assign full     = full_in ^ full_out;
   assign complete = !empty &&
          ( full || head_word != tail_nibble[2:1] );

   always @( posedge inclk or posedge reset )
     if( reset ) begin
        sto[7:0] = 0; tail_nibble = 0; empty_in = 1; full_in = 0;
     end else if( !full ) begin
        if( empty ) begin empty_in = !empty_in; tail_nibble = 0; end

        case ( tail_nibble )
          0: sto[7:0] = {4'b0,in};
          1: sto[7:4] = in;
          2: sto[15:8] = {4'b0,in};
          3: sto[15:12] = in;
          4: sto[23:16] = {4'b0,in};
          5: sto[23:20] = in;
          6: sto[31:24] = {4'b0,in};
          7: sto[31:28] = in;
        endcase

        tail_nibble = tail_nibble + 1;

        if( tail_nibble[0]==0 && tail_nibble>>1 == head_word )
          full_in = !full_in;

     end

   always @( posedge outclk or posedge reset )
     if( reset ) begin
        head_word = 0; empty_out = 0; full_out = 0;
     end else
       if( !empty ) begin
          if( full ) begin
             full_out = !full_out;
             head_word = head_word + 1;
          end else if( head_word == ( 3 & (tail_nibble-1)>>1 ) ) begin
             empty_out = !empty_out;
             head_word = 0;
          end else begin
             head_word = head_word + 1;
```

```
          end
        end

endmodule // width_change

// exemplar translate_off

module wc_test();

   // If non-zero, stop simulation when an error is encountered.
   // If zero, when an error is encountered simulation will proceed
   // and a count of errors will be displayed.
   parameter stop_on_err = 1;

   wire [7:0] out;
   wire       full, empty, comp;

   reg        outclk, inclk, reset;
   reg [3:0]  indata;

   reg [31:0] shadow;
   reg        shadow_full, shadow_comp, shadow_empty;
   reg [7:0]  shadow_head;
   integer    shadow_occ;
   reg        check;
   integer    phasecount;
   time       remove_delay_limit_short, remove_delay_limit_long;
   time       remove_delay_limit;
   time       fill_delay_limit;
   time       next_empty;

   integer    allow_simultaneous_clocks;
   integer    allow_overlapping_clocks;
   integer    error_out_t, error_out;
   integer    error_empty_t, error_empty;
   integer    error_comp_t, error_comp;
   integer    error_full_t, error_full;
   integer    error_test, error_test_t;

   width_change wc(out,full,comp,empty,outclk,indata,inclk,reset);


   function [31:0] randi;
      input [31:0] limit;
      randi = ( $random >> 1 ) % limit;
   endfunction // randi

   initial begin
      indata = 0; outclk = 0; inclk = 0; reset = 0;
      shadow_empty = 1; shadow_comp = 0; shadow_full = 0; shadow_head = 0;
      check = 0; shadow = 0; shadow_occ = 0; phasecount = 0;

      fill_delay_limit = 20;

      remove_delay_limit_short = 0.5 * fill_delay_limit * 2;
      remove_delay_limit_long = 2 * fill_delay_limit * 2;
      // Start filling.
      remove_delay_limit = remove_delay_limit_long;

      allow_overlapping_clocks = 0;
      allow_simultaneous_clocks = 0;
      error_out = 0; error_out_t = 0;
      error_empty = 0; error_empty_t = 0;
      error_comp = 0; error_comp_t = 0;
      error_full = 0; error_full_t = 0;
```

```
error_test = 0; error_test_t = 0;

reset = 1; #10 reset = 0; #10;

fork:TESTLOOP
   forever begin:FILL
      integer clk_fall_delay;
      integer clk_rise_delay;

      clk_fall_delay = allow_overlapping_clocks ?
                       randi(fill_delay_limit)+1 : 1;
      clk_rise_delay = randi(fill_delay_limit)+1;

      indata <= #(10*randi(clk_rise_delay)) $random;
      #(10*clk_rise_delay);

      // Special case: if FIFO is full or empty and data is
      // simultaneously clocked in and out there's no way to
      // tell if FIFO rejected the data begin clocked in.

      while( next_empty == $time &&
             ( !allow_simultaneous_clocks ||
               shadow_occ >= 20 || shadow_occ < 12 )) #10;

      inclk = 1;
      check <= #1 !check;
      shadow_empty = 0;

      if( !shadow_full ) begin
         shadow_occ = shadow_occ + 4;
         shadow = { indata, shadow[31:4] };
         if( shadow_occ > 28 )
           begin
              shadow_full = 1;
              if( remove_delay_limit === remove_delay_limit_long ) begin
                 remove_delay_limit = remove_delay_limit_short;
                 phasecount = phasecount + 1;
              end
           end
         if( shadow_occ > 7 ) shadow_comp = 1;
      end

      shadow_head = shadow >> ( 32 - shadow_occ );

      indata <= #(10*randi(clk_fall_delay)) $random;
      inclk = #(10*clk_fall_delay) 0;

   end

   forever begin:EMPTY
      integer clk_fall_delay;
      integer clk_rise_delay;

      clk_fall_delay = allow_overlapping_clocks ?
                       randi(remove_delay_limit)+1 : 1;
      clk_rise_delay = randi(remove_delay_limit)+1;

      next_empty = $time + 10 * clk_rise_delay;
      outclk = #(10*clk_rise_delay) 1;
      check <= #1 !check;
      shadow_full = 0;

      if( !shadow_empty ) begin
         if( shadow_occ <= 8 ) begin
            shadow_occ = 0; shadow_empty = 1;
```

```
            remove_delay_limit = remove_delay_limit_long;
         end else begin
            shadow_occ = shadow_occ - 8;
         end
         shadow_head = shadow >> ( 32 - shadow_occ );
         if( shadow_occ < 8 ) shadow_comp = 0;
      end
      outclk = #(10*clk_fall_delay) 0;
   end // block: EMPTY

   forever @( out or check ) #1
     if( out !== shadow_head ) begin
        if( stop_on_err ) begin
           $display("Wrong output.");
           #2 $stop;
        end
        error_out = error_out + 1;
     end

   forever @( empty or check ) #1
     if( empty !== shadow_empty ) begin
        if( stop_on_err ) begin
           $display("Wrong empty.");
           #2 $stop;
        end
        error_empty = error_empty + 1;
     end

   forever @( comp or check ) #1
     if( comp !== shadow_comp ) begin
        if( stop_on_err ) begin
           $display("Wrong complete.");
           #2 $stop;
        end
        error_comp = error_comp + 1;
     end

   forever @( full or check ) #1
     if( full !== shadow_full ) begin
        if( stop_on_err ) begin
           $display("Wrong full.");
           #2 $stop;
        end
        error_full = error_full + 1;
     end

   forever @( phasecount ) begin:P
      reg [84:0] test_name;
      if( phasecount == 200 || phasecount == 400 || phasecount == 600 )
        begin

           error_test = error_out + error_empty + error_comp + error_full;
           error_test_t = error_test_t + error_test;
           error_out_t   = error_out_t + error_out;
           error_empty_t = error_empty_t + error_empty;
           error_comp_t  = error_comp_t + error_comp;
           error_full_t  = error_full_t + error_full;

           case( phasecount )
             200:test_name = "No Overlap";
             400:test_name = "Not Simult";
             600:test_name = "Full Test";
           endcase // case( phasecount )

           $display("Test %s.  Total Errors %d. Errors by type:",
```

```
                    test_name, error_test);
                $display("         Output %d, empty %d, compl %d, full %d",
                    error_out, error_empty, error_comp, error_full);


                if( phasecount == 200 ) begin
                   allow_overlapping_clocks = 1;
                   if( error_test === 0 )
                     $display("Passed all non-overlapping tests!!");
                end else if ( phasecount == 400 ) begin
                   allow_simultaneous_clocks = 1;
                   if( error_test === 0 )
                     $display("Passed all overlapping-but-not-simultaneous tests!!!!!");
                end else if ( phasecount == 600 ) begin
                   if( error_test === 0 )
                     $display("Passed all simultaneous clock tests!!!!!!!");
                   if( error_test_t === 0 )
                     $display("Passed EVERY test!  PERFECT!!!!!!!!!!!!!!!");
                   else
                     $display("Failed %d tests. :-(",error_test_t);
                   disable TESTLOOP;
                end

                error_out = 0; error_empty = 0; error_comp = 0; error_full = 0;
                error_test = 0;

            end // if ( phasecount == 200 || phasecount == 400 || phasecount == 600 )
        end
      join

   end // initial begin

endmodule // wc_test

// exemplar translate_on
```