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.

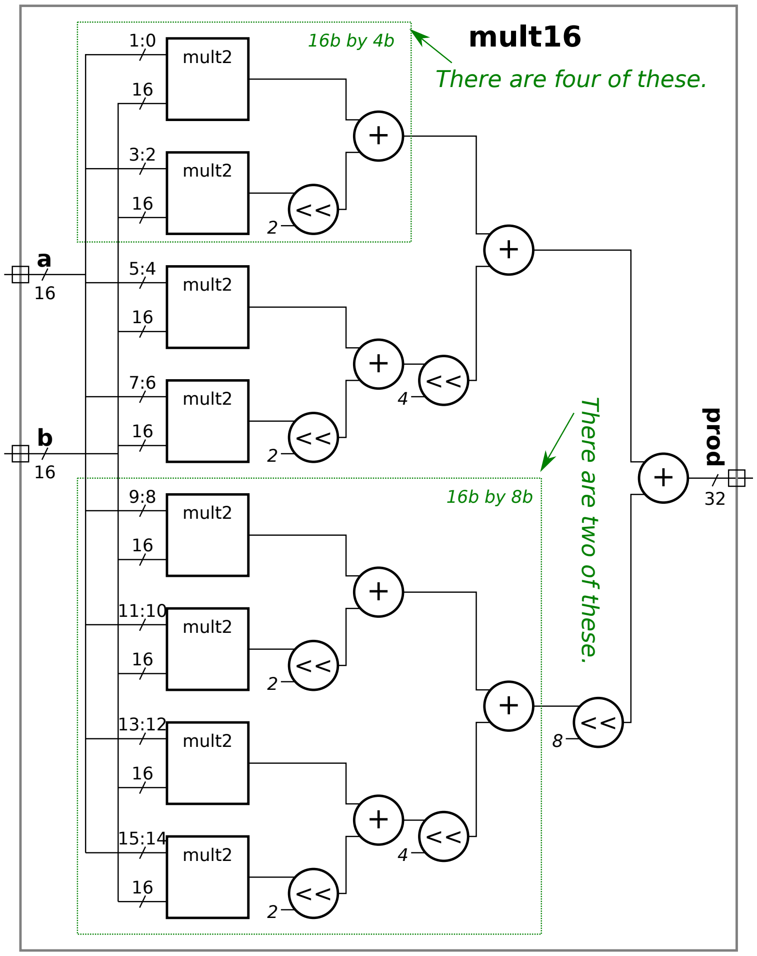
Synthesis

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 $0.1\,\text{ns}$ delay target and least expensive with the $10\,\text{ns}$ 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   Delay
                                             Actual  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.