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 $\boxed{\texttt{F9}}$ 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 $\boxed{\texttt{F9}}$).

```
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
```