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
```