Name Solution

Digital Design Using HDLs
LSU EE 4755
Solve-Home Midterm Examination

Friday, 6 Nov 2020 to early Monday, 9 Nov 2020 05:00 CST)

Work on this exam alone. Regular class resources, such as notes, papers, documentation, and code, can be used to find solutions. Outside material that covers the same topics, such as Verilog tutorials, digital logic design guides can also be used. Do not try to directly seek out solutions to any question here. That is, don't Web-search the text of a problem. Do not discuss this exam with classmates or anyone else, except questions or concerns about problems should be directed to Dr. Koppelman.

Warning: Unlike homework assignments collaboration is not allowed on exams. Suspected copying will be reported to the dean of students. The kind of copying on a homework assignment that would result in a comment like "See ee4755xx for grading comments" will be reported if it occurs on an exam. Please do not take advantage of pandemic-forced test conditions to cheat!

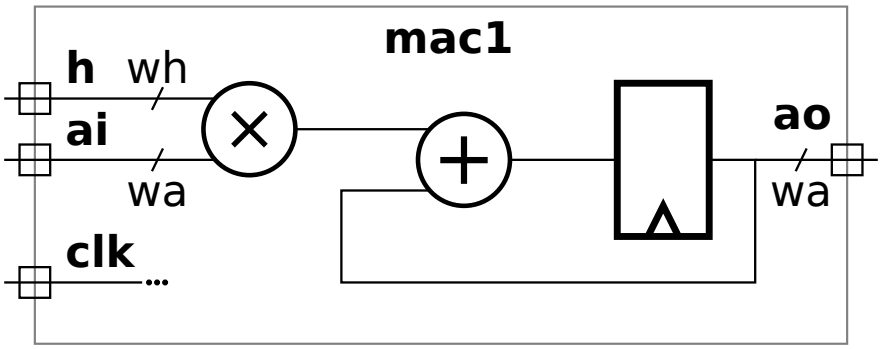
Problem 1	_____	(20 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(20 pts)
Problem 5	_____	(20 pts)
Exam Total	_____	(100 pts)

☣ $r \geq 2\,\mathrm{m} \quad \Rightarrow \quad R_e < 1$

Good Luck!

Problem 1: [20 pts] Appearing below are some variations on a multiply accumulate module.

(*a*) Complete the Verilog code below so that it matches the illustration.



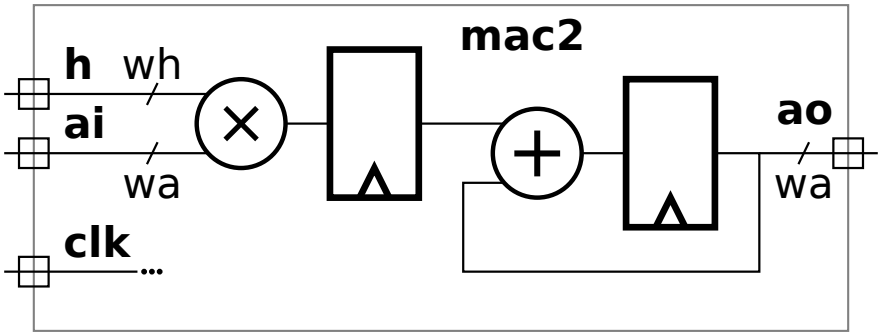
☑ Complete the Verilog.

☑ Use parameters for the bit widths `wh` and `wa`.

☑ The registers inferred from the Verilog must match the diagram.

```
 /// SOLUTION
module mac1
  #( int wa = 32, wh = 16 )
   ( output logic [wa-1:0] ao,
     input uwire [wh-1:0] h,
     input uwire [wa-1:0] ai,
     input uwire clk );

   always_ff @( posedge clk ) ao <= h * ai + ao;

endmodule
```

(*b*) Complete the Verilog code below so that it matches the illustration, similar to the one on the previous page.



☑ Complete the Verilog.

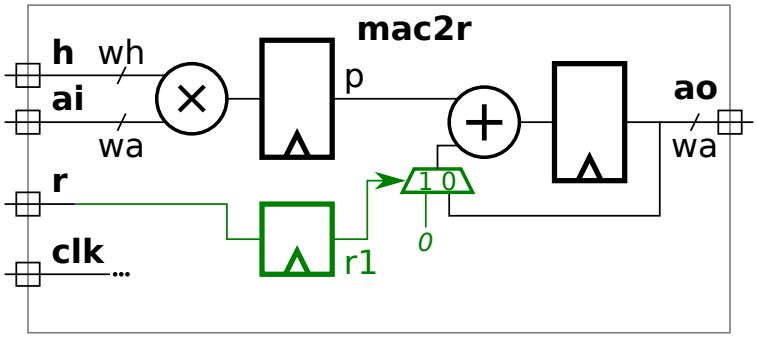
☑ Use parameters for the bit widths `wh` and `wa`.

☑ The registers inferred from the Verilog must match the diagram.

```
 /// SOLUTION
module mac2
   #( int wh = 4, wa = 3 )
   ( output logic [wa-1:0] ao,
     input uwire [wh-1:0] h,
     input uwire [wa-1:0] ai,
     input uwire clk );

   logic [wa-1:0] p;

   always_ff @( posedge clk ) begin
      p <= h * ai;
      ao <= p + ao;
   end

endmodule
```

Problem 2: [20 pts] The mac (multiply-accumulate) modules compute a running sum of products. The alert student might have noticed that there is no way to reset the sum. In this problem a reset will be added.

The module below has an input `r` (for reset) which is to work as follows: When `r=1` at a positive edge the product `h*ai` should start a new running sum. That is, that particular `h*ai` should be added to zero. When `r=0` at a positive edge the product `h*ai` should be added to the sum of the previous products. (If `r=0` is always true then the hardware as illustrated works correctly.)



☑ Add hardware to the diagram to implement the reset. ☑ Complete the Verilog to implement the reset.

☑ Use parameters for the bit widths `wh` and `wa`.

☑ The registers inferred from the Verilog must match the diagram and ☑ be sure that the reset is applied to the correct value.

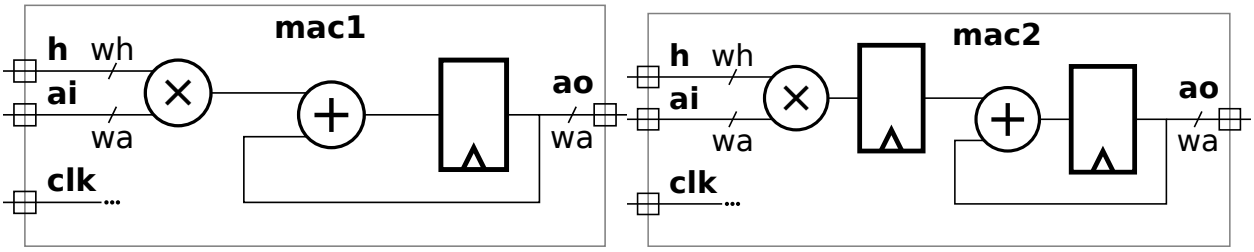
The hardware changes appear above in green and the Verilog code appears below in all sorts of colors.

The problem states that when `r=1` the accompanying values of `h` and `ai` must start a new running sum. To implement this a register has been added, `r1`, so that the value of `r` moves with the product `h*ai`, so that in the next cycle that product `h*ai` is added to zero rather than to `ao`. If `r` were connected directly to the multiplexor then the `h*ai` arriving with `r` would be added to a non-zero value.

Grading Note: No one solved this 100% correctly.

```
/// SOLUTION
module mac2r #( int wh = 4, wa = 3 )
   ( output logic [wa-1:0] ao,
     input uwire [wh-1:0] h,
     input uwire [wa-1:0] ai,
     input uwire r, clk );

   logic [wa-1:0] p;
   logic r1;

   always_ff @( posedge clk ) begin
      r1 <= r;
      p <= h * ai;
      ao <= p + ( r1 ? 0 : ao );
   end
endmodule
```

Problem 3: [20 pts] Appearing below are the modules from the previous problem. Suppose that in the multiplier below bit i of the product were computed in time $[4i+2]\,\mathrm{u_t}$ and that a ripple adder were used for the sum. Let w denote the value of `wh` and `wa` (which means `wh==wa`).



(*a*) Find the minimum clock period for each using the simple model, and taking into account cascading. (The clock period is the length of the critical path, including the register delay.)

☑ Find the clock period for `mac1` with cascading. ☑ Don't forget to include the delay of the register.

Short answer: The clock period is $[4(w-1)+2+4+6]\,\mathrm{u_t} = [4w+8]\,\mathrm{u_t}$.

Explanation: Taking into account cascading in this case means that when we compute the time needed to compute the addition we take into account the fact that bit i arrives at time $4i+2$ and that a ripple adder is used to compute the sum. Bit 0 of the product arrives at the adder's bit 0 BFA at time $4\times 0+2=2$, and so its outputs, sum and carry-out, arrive at $2+4=6$. Bit 1 of the product arrives at the adder's bit 1 BFA at time $4\times 1+2=6$ as does the carry out from bit 0, so the sum and carry out won't be available until $6+4=10$. This pattern persists, so bit i of the adder output is available at time $4i+2+4=4i+6$. The adder is w bits wide, so the MSB is not available until time $[4(w-1)+6]\,\mathrm{u_t} = [4w+2]\,\mathrm{u_t}$. To compute the clock period we need to tack on the $6\,\mathrm{u_t}$ register delay, bringing the clock period to $[4w+2+6]\,\mathrm{u_t} = [4w+8]\,\mathrm{u_t}$.

☑ Find the clock period for `mac2` with cascading. ☑ Don't forget to include the delay of the registers.

Short answer: The clock period is $[\max\{\,4(w-1)+2,\,2(w+1)\,\}+6]\,\mathrm{u_t} = [\max\{\,4w-2,\,2w+2\,\}+6]\,\mathrm{u_t} = [4w-2+6]\,\mathrm{u_t} = [4w+4]\,\mathrm{u_t}$, for $w \geq 2$.

Explanation: The clock period is determined by the critical (longest) path. Paths start at launch points and end at capture points. Register outputs are launch points and register inputs (both data and enable) and capture points. Usually (but not always) module inputs are launch points and module outputs are capture points. There are two possible critical paths. Path one is from `h` (or `ai`), through the multiplier, to the register input, path two is from `ao`, through the adder, to the register input. The length of path one is $4(w-1)+2+6=4w+4$ and the length of path two is $2(w+1)+6=2w+8$. When $w \geq 1$ path one is longer and so the clock period must be $[4w+4]\,\mathrm{u_t}$.

And what about cascading? That doesn't apply here because there is a register between the multiplier and the adder and so all bits arrive at the input to the adder at the same time.

(*b*) Find the minimum clock period for each using the simple model assuming that the multiplier output and adder input could not cascade.

☑ Find the clock period for `mac1` without cascading. ☑ Don't forget to include the delay of the register.

Short Answer: The clock period is $[4(w-1)+2+2(w+1)+6]\,\mathrm{u_t} = [6w+6]\,\mathrm{u_t}$.

Explanation: Without cascading the adder must wait for every bit of the product to be computed. The last bit of the product is available at $4(w-1)+2$ and only then can the addition start (with the no-cascading assumption). So adding the addition time, $2(w+1)$, and register delay, 6, gives the clock period.

Note that the no-cascading assumption was made for pedagogical reasons. If indeed bit i of the product arrives at $4i+2$ and a ripple adder is used, cascading should be taken into account when computing the delay.

☑ Find the clock period for `mac2` without cascading. ☑ Don't forget to include the delay of the registers.

The clock period for `mac2` is the same with and without the cascading assumption, so the period is the same as the one computed above, $[4w + 4]\,\mathrm{u_t}$.

Problem 4: [20 pts] Appearing below is a recursively defined multiplier constructed using bfa (binary full adder) and bha (binary half adder) modules.

```
module mult_tree_bfas #( int wa = 16, int wb = wa, int wp = wa + wb )
   ( output uwire [wp-1:0] prod, input uwire [wa-1:0] a, input uwire [wb-1:0] b );

   if ( wa == 1 ) begin
      assign prod = a ? b : 0;
   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-1:0] prod_lo, prod_hi;

      mult_tree_bfas #(wn,wb) mlo( prod_lo, a[wn-1:0],  b );
      mult_tree_bfas #(wn,wb) mhi( prod_hi, a[wa-1:wn], b );

      assign prod[wn-1:0] = prod_lo[wn-1:0];

      uwire c[wp-1:wn-1];
      assign c[wn-1] = 0;
      for ( genvar i=wn; i<wx; i++ )
        bfa b(c[i], prod[i], prod_lo[i], prod_hi[i-wn], c[i-1] );
      for ( genvar i=wx; i<wx+wn; i++ )
        bha b(c[i], prod[i], prod_hi[i-wn], c[i-1] );
      localparam int wz = wp - wx - wn;
      if ( wz > 0 ) assign prod[wp-1 :- wz] = 0;
   end
endmodule
```

Show the hardware that will be inferred for two levels of recursion and compute its cost. That is, show three instances of `mult_tree_bfas`: a top-level one, and two recursive instantiations. Show the hardware for the top-level instance and both of the two recursive instantiations. (It is only necessary to show two levels.) Do this for `wa=8` in the top-level module.

Continued on next page.

☑ Show the inferred hardware.

The inferred hardware is shown on the next page. The binary full and half adders are shown as boxes. Only the carry-out port is labeled, with a `co`, of course. As most reading this should easily figure out, the port at the top of each BFA and BHA module is carry-in, the port on the right-hand side is the sum, and the two BFA inputs on the left hand side are the two bits to be added together.

Note that the bits of `a` are split so that the less significant bits connect to `mlo` and the more significant bits connect to `mhi`. In contrast *all bits of* `b` connect to both `mlo` and `mhi`.

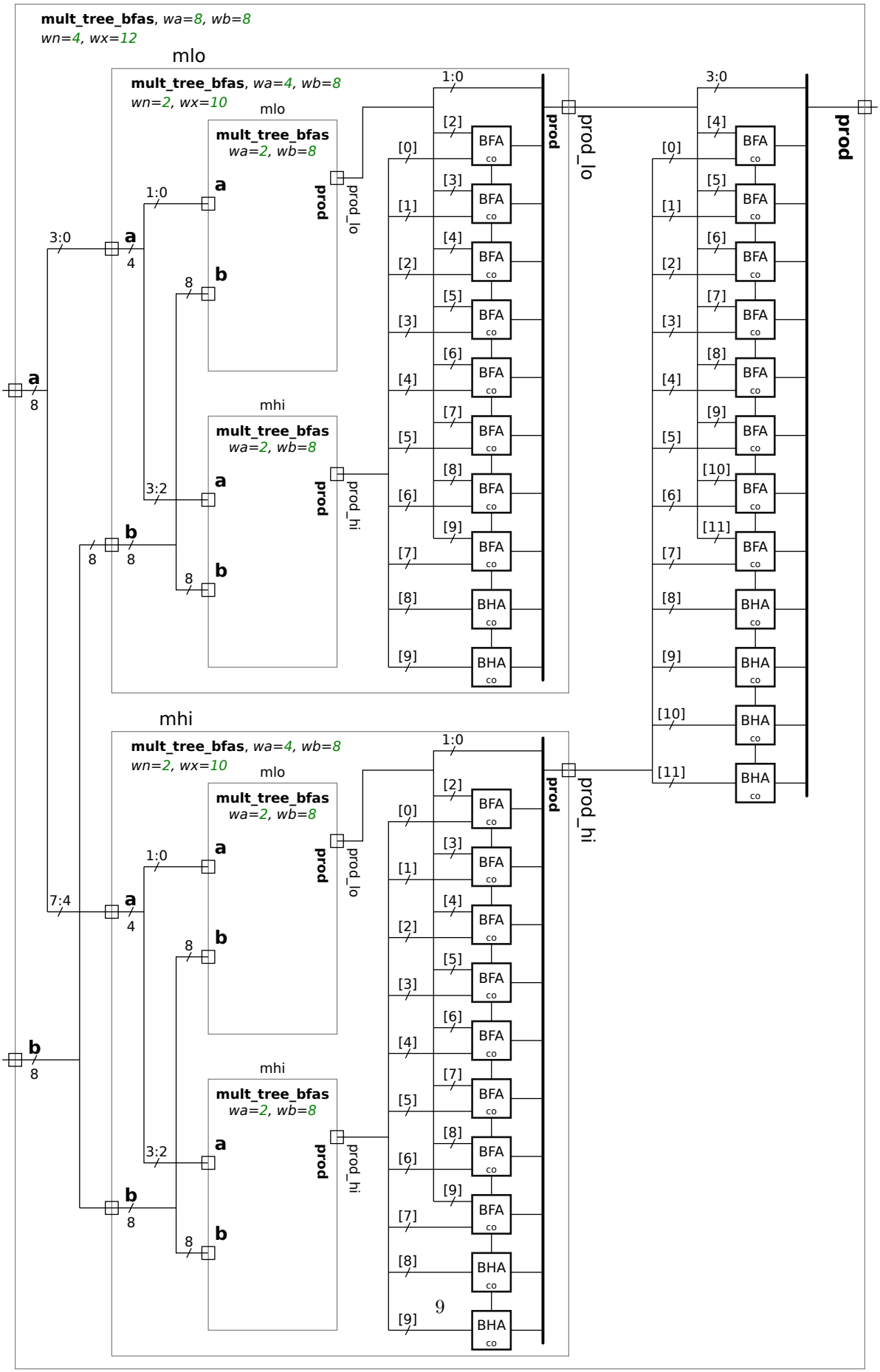
☑ Be sure to distinguish hardware (such as a `bfa` module) from values computed during elaboration.

An example of a value computed during elaboration is `wx`. The value at each level is shown. Since the value has been computed during elaboration there is no need to emit hardware to compute a value that is already known.

☑ Compute the cost of the hardware in your diagram using the simple model. (Work out the cost of a bha by hand.) ☑ The cost should be for two levels, not for hardware going down to the base case.

As can be seen by looking at the loop bounds of the generate loops, each instance consists of `wb` BFA modules and `wn` BHA modules. For the top-level (`wa=8`) instance `wb=8` and `wn=4`. In the `mlo` and `mhi` instances instantiated in the top level `wb=8` and `wn=2`. (Yes, `wb` is 8 at every level.) So the number of BFA modules is $8 + 2 \times 8 = 24$, and the number of BHA modules is $4 + 2 \times 2 = 8$. The cost of a BFA is $9\,\mathrm{u_c}$. A BHA can be constructed with an XOR gate for the sum and an AND gate for the carry out, for a cost of $4\,\mathrm{u_c}$. However the carry out can be used to compute the sum: `s = ( a || ci ) & !co` where `co = a && ci`. Such a construction has a cost of just $3\,\mathrm{u_c}$.

The total cost is $[24 \times 9 + 8 \times 3]\,\mathrm{u_c} = 240\,\mathrm{u_c}$ using the $3\,\mathrm{u_c}$ BHA or $[24 \times 9 + 8 \times 4]\,\mathrm{u_c} = 248\,\mathrm{u_c}$ using the $4\,\mathrm{u_c}$ BHA.


mult_tree_bfas, wa=8, wb=8
wn=4, wx=12
mlo
mult_tree_bfas, wa=4, wb=8
wn=2, wx=10
mlo
mult_tree_bfas
wa=2, wb=8
mhi
mult_tree_bfas
wa=2, wb=8
mhi
mult_tree_bfas, wa=4, wb=8
wn=2, wx=10
mlo
mult_tree_bfas
wa=2, wb=8
mhi
mult_tree_bfas
wa=2, wb=8
BFA
BHA
co
prod
prod_lo
prod_hi
a
b
3:0
7:4
1:0
3:2

Problem 5: [20 pts] Answer each question below.

(*a*) Appearing below is a multiply/add module, `nnMADDfp`, that computes its result using a FP add and multiply module. The values on the ports are IEEE 754 floats, and when `wa=32` the format is IEEE 754 single, the same as a SystemVerilog `shortreal`. That is followed by an incomplete testbench module, `testnnMADD`. The testbench module generates random values for the `nnMADDfp` module in variables `ar`, `br`, and `sir`, and computes what the result should be, `sor`.

Add Verilog code to deliver `ar`, `br`, and `sir` to the `nnMADDfp` instance, and to put the output of `nnMADDfp` into `sor_mut` so that `sor_mut` has the correct type of value. Note that one does not need to understand what is inside of `nnMADDfp`, `nnAddfp`, nor `nnMultfp`.

☑ Deliver (whatever that means) `ar`, `br`, and `sir` to `nnMADDfp` instance. ☑ Get output of the `nnMADDfp` instance into variable `sor_mut`.

The solution is shown below. First the inputs to instance `n` of `nnMADDfp`, variables `a`, `b`, and `si`, must be assigned the values in variables `ar`, `br`, and `sir`. Because `a`, `b`, and `si` are of type `logic` a statement like `a=ar` won't work because for such a statement Verilog will first convert `ar` from a `shortreal` to a 32-bit unsigned integer (`logic [31:0]`). It won't work because module `nnMADDfp` expects `a`, though declared `logic`, to be in the same format as `shortreal`. To avoid the problem the Verilog system task `$shortrealtobits` is used. That avoids the `shortreal`-to-integer or any other conversion. The bits are left unchanged. A similar function is used for re-interpreting the module output, `so`, from `logic` to `shortreal`.

A serious error which too many students made was instantiating an `nnMADDfp` module inside the `t` loop. First, the module is already instantiated. Second, it makes no sense to instantiate a module in procedural code.

Note: In the original exam `ar`, `br`, etc. were declared `real` instead of `shortreal`. The solution would be no different: `$shortrealtobits` should still be used. But the explanation above would have been more complicated since in statement `a=$shortrealtobits(ar)` there would be a conversion: `ar` would be converted from `real` to `shortreal`, but then the bits in the `shortreal` would be assigned to `a` with no further changes.

Verilog code, including the solution, on next page.

```
module nnMADDfp #( int wa = 10 )
  ( output uwire [wa-1:0] so, input uwire [wa-1:0] a, b, si);
   uwire [wa-1:0] p;
   nnMultfp #(wa) mu(p, a, b);
   nnAddfp #(wa) ad(so, si, p);
endmodule

module testnnMADD;
   localparam int w = 32, ntests = 100;
   uwire [w-1:0] so;
   logic [w-1:0] a, b, si;
   nnMADDfp #(w) n(so, a, b, si);

   initial begin

      for ( int t=0; t<ntests; t++ ) begin
         shortreal sor, ar, br, sir, sor_mut;
         ar = rand_fp();    // Value to be used as input a to nnMADDfp.
         br = rand_fp();    // Value to be used as input b to nnMADDfp.
         sir = rand_fp();   // Value to be used as input si to nnMADDfp.
         sor = ar * br + sir;

         /// SOLUTION
         a = $shortrealtobits(ar);   // Move bits of ar to a without changing them.
         b = $shortrealtobits(br);   // This operation is sometimes called ..
         si = $shortrealtobits(sir); // .. a reinterpretation cast.

         #1;

         sor_mut = $bitstoshortreal(so);  // <-- MORE OF THE SOLUTION.


         if ( sor != sor_mut ) handle_incorrect_result();

      end
   end
endmodule
```

(*b*) The module below will not compile or simulate due to multiple assignments to `temperature`, which is declared `uwire`. Changing `uwire` to `wire` will fix the compile problem. Nevertheless, is that the right fix?

```
module more_stuff #( int w = 16 )
   ( output uwire [w-1:0] v, y, input uwire [w-1:0] a, b, c );

   uwire [w-1:0] temperature;

   assign temperature = a + b;
   assign v = temperature >> c;
   assign temperature = a - b;
   assign y = temperature << c;

endmodule
```

```
module more_stuff #( int w = 16 )
   ( output uwire [w-1:0] v, y, input uwire [w-1:0] a, b, c );

   /// SOLUTION

   uwire [w-1:0] t1, t2;

   assign t1 = a + b;
   assign v = t1 >> c;
   assign t2 = a - b;
   assign y = t2 << c;

endmodule
```

☑ What problem remains after changing `temperature` from a `uwire` to a `wire`?

Short answer: The same value of `temperature` is used to compute both `v` and `y`, though the coder's intent may have been different values. That value of `temperature` consists of bits common to `a+b` and `a-b`, and `x`'s elsewhere. It's not likely the coder intended that either.

Longer explanation: With the "fix" object `temperature` is driven by two different assignments, `a+b` and `a-b`. In bit positions where `a+b` and `a-b` are both `0`, the value would be `0`. In bit positions where `a+b` and `a-b` are both `1`, the value would be `1`. But, in bit positions where `a+b` and `a-b` differ the value would be `x`.

An important thing to remember is that continuous assignments, which is what the `assign` keyword specifies, are executed whenever objects on the right-hand side change. As a consequence the values for both `v` and `y` will ultimately be computed with **the same value** of `temperature`.

☑ Fix the problem based on what the code looks like its trying to do.

Solution appears above.

(*c*) An important part of synthesis is optimizing. It is possible to optimize before and again after technology mapping.

☑ What is technology mapping? ☑ Show an example of logic before and after technology mapping. (Make up some technology.)

In the technology mapping step generic gates are replaced with gates in the target technology. For example, consider the expression `y=!( a & b || c & d )`. That might be inferred into the following generic gates: two AND gates and one NOR gate. A target technology might have a special AND-OR-INVERT gate that computes the entire expression, and because of the way CMOS FETs can be interconnected does so using less time or area than two AND gates and a NOR gate in the same technology.

☑ Describe an optimization that can be done before technology mapping. Provide an example. (This is done all the time in class.)

Expression `a || (!a) && b` can be optimized to `a || b`. Other examples include constant propagation and folding, such as `a && 1` being optimized to `a`.

☑ Describe an optimization that can be done only after technology mapping (or perhaps during). Provide an example, feel free to make things up.

Realistic timing data is available for the gates used in technology mapping. For that reason, any optimization that reduces delay should be done after technology mapping.