Name Solution

Digital Design Using HDLs
LSU EE 4755
Midterm Examination

Friday, 27 October 2023, 11:30-12:20 CDT

Problem 1		(30 pts)
Problem 2		(25 pts)
Problem 3		(30 pts)
Problem 4		(15 pts)
Exam Total		(100 pts)

Alias Again on 8 April!

Good Luck!

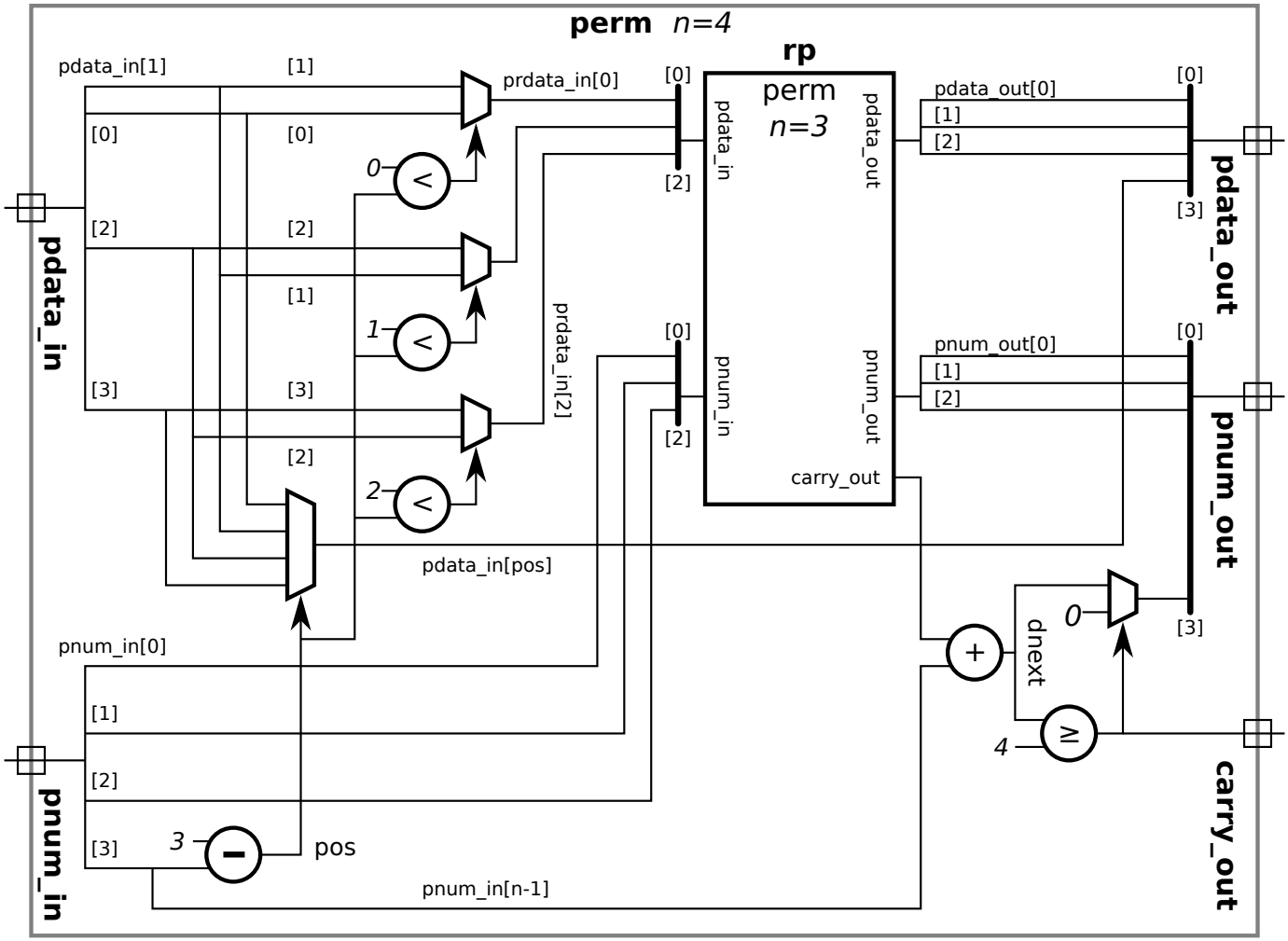
Problem 1: [30 pts] Appearing below is the permutation module from the solution to Homework 3. Using the illustration of the ports show the inferred hardware for an instantiation with `n=4`. Show the `n=4` instantiation but not what is inside the `n=3` recursive instantiation.

```
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

      uwire [wd-1:0] pos = n - 1 - pnum_in[n-1];
      assign pdata_out[n-1] = pdata_in[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,wd) rp( pdata_out[0:n-2], pnum_out[0:n-2], co,
                           prdata_in, pnum_in[0:n-2] );

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

   end

endmodule
```

☑ Show inferred hardware for `n=4`. Be sure to use ☑ the illustrated module ports and to show ☑ the recursively instantiated module (but not its contents).

☑ Show hardware, ☑ do not confuse elaboration-time computation with computation hardware.

Solution appears below. Notice that elaboration-time constants such as `i` and `n` are replaced by their values. Notice also that `pdata_in[i]` for `i=0` is inferred simply as a wire connecting to input `pdata_in[0]`, whereas `pdata_in[pos]`, because `pos` is not a constant, is inferred as a multiplexor with data inputs connecting to each `pdata_in` input.



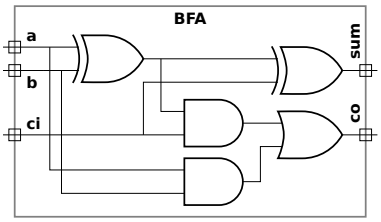
Problem 2: [25 pts] A ripple adder to compute $a + b$ is to be used in situations where a is a constant.

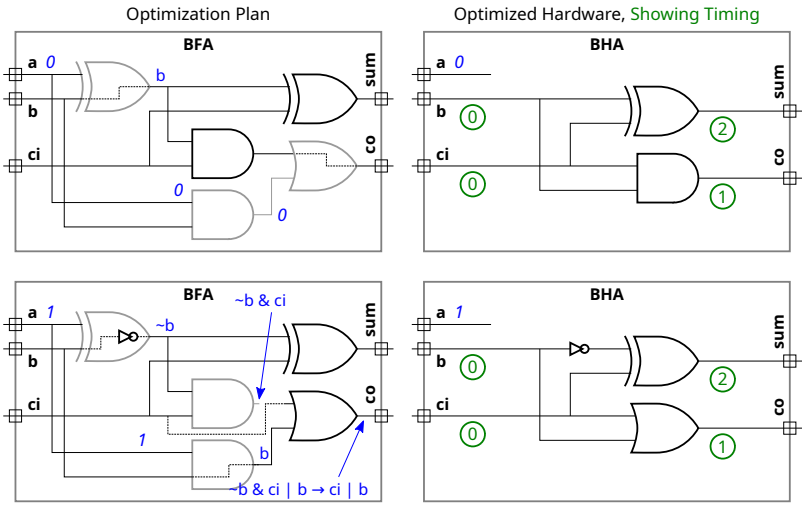
(a) Find the cost and delay of a BFA with input a constant (for use in the ripple adder). A BFA is shown for your convenience.

☑ Show the BFA(s) optimized for input a constant.

☐ Use a truth table to find optimizations not revealed by constant pushing: in a correct solution the delay does not depend upon a.

☑ Show simple-model cost of this(these) module(s) and ☑ show simple-model delay(s) of this(these) module(s).





The optimized BFAs appear on the right. The first row shows optimization for a=0 and the second row for a=1. The first column, headed Optimization Plan, shows the impact of the constant a value on gates in the full BFA, the second column shows a cleaned-up version of the logic, and, for the timing analysis, with arrival times labeled in green.

A binary full adder with one constant input is little different than a binary half adder, and so a constant-input BFA is labeled BHA. The optimization for the a=0 case is straightforward. For the a=1 case an initial optimization would use both an AND gate and an OR gate to compute co. But the logic can be simplified further by noting that when b=1 directly connecting ci to the OR gate has no effect, and when b=0 directly connecting ci to the OR gate has the intended effect. Or perhaps one just remembers the Boolean algebra identity $x + \overline{x}y = x + y$.

The cost of each module is $3 + 1 = 4\,\mathrm{u_c}$. Actually, the cost of each module can be reduced to just $3\,\mathrm{u_c}$ by splitting the XOR into three gates and using the AND or OR gate to replace one of those gates with other zero-cost changes needed to compute exclusive or. Final exam problem?

The arrival times are labeled in green. In both modules the delay of sum is $2\,\mathrm{u_t}$ and the delay of co is $1\,\mathrm{u_t}$. Lucky for us the delay of co is $1\,\mathrm{u_t}$, because that impacts the delay of the ripple adder.

(b) On the facing page show the optimized hardware, cost, LSB delay, and MSB delay of a w-bit ripple adder for computing $a + b + c_{\mathrm{in}}$, where c_{in} is a carry-in bit (cin in the diagram) and a is a constant. (☑ See the check box items for details.) Use the illustration on the facing page as a starting point.

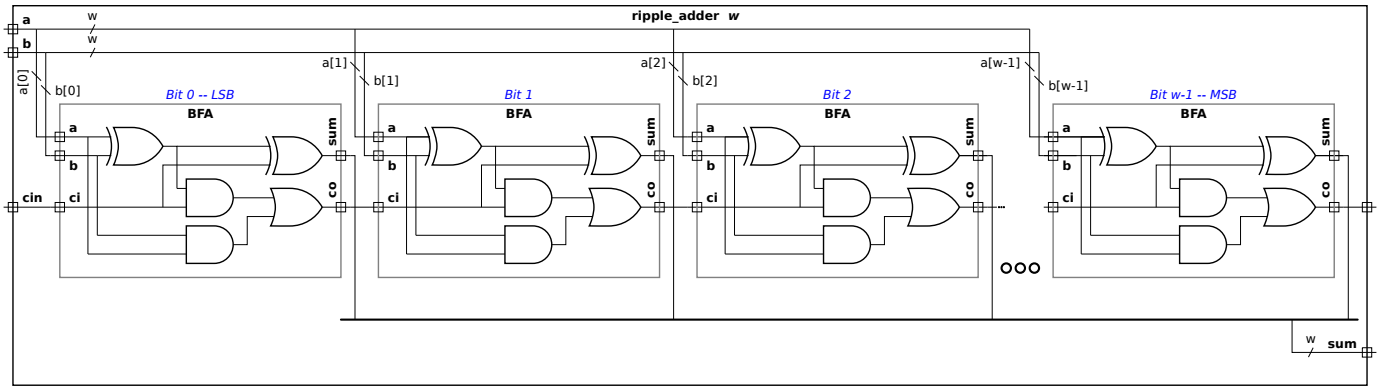
☑ Show the hardware optimized for a constant a and a non-constant cin.

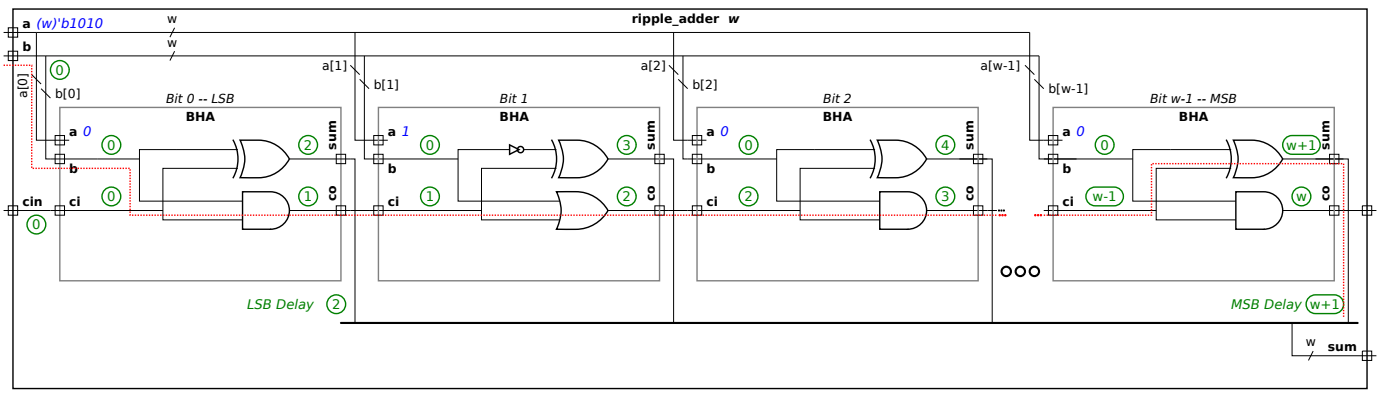
☑ Compute the simple-model cost of this hardware in terms of w.

☑ Compute the simple-model delay of the LSB of the sum.

☑ Compute the simple-model delay of the MSB of the sum in terms of w and ☑ show the critical path.

☑ Don't forget that a is a constant.





Solution to part b appears above. The exact BHA units to use depend on the value of a. The diagram is for $a = 1010_2$.

The cost of each BHA is $4\,\mathrm{u_c}$, so $\boxed{\text{the total cost is } 4w\,\mathrm{u_c}}$. Based on the analysis shown in green on the diagram the $\boxed{\text{LSB delay is } 2\,\mathrm{u_t}}$ and the $\boxed{\text{MSB delay is } [w+1]\,\mathrm{u_t}}$. The critical path for the MSB is shown in red.

(c) If `cin` were removed (or set to zero) the cost and delay of the optimized adder would depend on `a`. Explain why, and illustrate with the example of `a=2`.

☑ How are cost and delay dependent on `a` when `cin` removed? ☑ Explain using the example `a=2`.

If `cin=0` the cost of the bit 0 BHA drops to zero. If bit `a[0]=0` then the `co` of the bit 0 BHA is 0, a constant, while if `a[0]=1` the `co` output is `b[0]`, not a constant. So for the case of $a = 10_2$ the `co` output of the bit 0 BHA is 0, but the `co` output of the Bit 1 BHA is `b[1]`, which is not a constant. For this $a = 10_2$ case the cost of the bit 0 and bit 1 BHAs is zero, but the cost of the remaining BHAs is $4\,\mathrm{u_c}$ each. For $a = 100_2$ the cost of the first three BHAs would be zero. So the cost of the constant adder with `cin=0` depends on the number of consecutive 0s starting at the LSB of `a`.

Problem 3: [30 pts] Answer the following Verilog questions.

(*a*) The module below makes extensive use of multidimensional arrays.

```
//                          2       1                      3    4       1    2
module mda( input uwire [2:1] c [5:1],    input uwire [7:1][2:1] a [5:1][3:1] );

   //   ☑   Add dimension(s) to the declaration of e so that the assignment is correct.
   //
   //     SOLUTION
   uwire    [2:1]    e            = c[1];


   //   ☑   Add dimension(s) to the declaration of b so that the assignment is correct.
   //
   //     SOLUTION
   uwire     [7:1][2:1]   b [3:1]  = a[1];


   logic g [7:0];
   logic [7:0] h;

   initial begin
//  ☑  Which is correct,  ◯ the assignment to g  or  ⊗ the assignment to h.  ☑  Explain.
      g = 1;  // Compile error because g is an unpacked array and 1 is a scalar.
      h = 1;  // Correct, h is a packed array and so is treated as an integer.
   end

endmodule
```

☑ What is the size of `c`, in bits? ☑ What is the size of `a`, in bits?

The size of object `c` is $2 \times 5 = 10$ bits. The size of object `a` is $7 \times 2 \times 5 \times 3 = 210$ bits.

(*b*) The module below does not compile.

```
module more_stuff #( int n = 20 ) ( output uwire [31:0] sum, input uwire [31:0] a [ n ] );
   logic [31:0] acc;
   always_comb begin
      acc = a[0];
      for ( int i=1; i<n; i++ )
        my_fixed_adder a1(acc, acc, a[i] );
   end
   assign sum = acc;
endmodule
```

☑ Describe the major problem. ☑ **DO NOT** try to fix the problem.

A module cannot be instantiated in procedural code, which the code above is doing with `my_fixed_adder`. This is a major problem because it can't be fixed by just changing a declaration or adding new objects. The instantiation must be removed from the procedural code. Another problem is that `acc` connects to two parts of `my_fixed_adder`. A reasonable guess would be that one of those is an output and the other is an input. It makes no sense to connect the same object to both an input and an output.

(c) The module below is supposed to set $x = a^2 + b^2$.

```
module wrong_way( output logic [31:0] x, input uwire [15:0] a, b );
   logic [31:0] asq;
   uwire [31:0] bsq = b * b;

   // initial asq = a * a;  // Original line.
   always_comb asq = a * a; // Corrected line.
   always_comb x = asq + bsq;

endmodule
```

☑ Explain the problem. ☑ Using sample inputs show the expected output and the actual output.

☑ Fix the problem.

Because **asq** is assigned in an **initial** block it will only be assigned once. Suppose at $t = 0$ the inputs are **a=2** and **b=3**. Then the correct output will appear, **x=13**. But suppose at $t = 1$ the inputs change to **a=5** and **b=6**. Object **asq** will keep its initial value, 4, and so the output will be $x = 2^2 + 6^2 = 40$ (computed using the $t = 0$ value of a and the $t = 1$ value of b).

The simplest fix is to change **initial** to **always_comb**, that's shown above.

(d) The module below does not compile.

```
module my_adder( output uwire [31:0] s, input uwire [31:0] a, b );
   always_comb s = a + b;
endmodule

module my_adder( output logic [31:0] s, input uwire [31:0] a, b );
   // Fixed module.
   always_comb s = a + b;
endmodule
```

☑ Why won't module above compile? ☑ Fix problem by changing declarations.

Because **s** is assigned in procedural code it must be a var kind, not a net kind. (A **uwire** is a net kind.) The fixed module is shown below the broken one.

(e) The module below compiles but does not provide the expected outputs, $p_a = a^2$, $p_b = b^2$, and $p = a^2 + b^2$.

```
module incorrect_way( output logic [31:0] pa,pb,p, input uwire [15:0] a, b );
   wire [31:0] sq;
   assign sq = a * a;
   always_comb pa = sq;
   assign sq = b * b;
   always_comb pb = sq;
   always_comb p = pa + pb;
endmodule
```

```
module correct_way( output logic [31:0] pa,pb,p, input uwire [15:0] a, b );
   /// SOLUTION
   uwire [31:0] sqa, sqb;
   assign sqa = a * a;
   always_comb pa = sqa;
   assign sqb = b * b;
   always_comb pb = sqb;
   always_comb p = pa + pb;
endmodule
```

☑ What will be the values of outputs `pa`, `pb`, and `p`?

If $a \neq b$ the value of each output will be `x`, the Verilog value indicating (in this case) conflicting drivers to an output. If $a = b$ then the correct result will be computed.

☑ Describe the problem. ☑ Fix it.

The problem is that `sq` is continuously assigned in **two** places, which though it does not violate any Verilog rules (note that `sq` is declared `wire` rather than `uwire`) it nevertheless does nothing useful. Suppose the `a*a` line drives `sq[0]` toward `0` and the `b*b` line drives bit `sq[0]` toward `1`. The value of `sq[0]` will be `x`, which is not what we want.

A simple fix is to use different objects for `a*a` and `b*b`. That is shown above.

Grading Note: Many students incorrectly described the value of `sq` as alternating between `a*a` and `b*b`. That doesn't happen because the simulator computes `sq` by first combining the `a*a` and `b*b` values (maybe using an old `a` or `b`, but always combining the two). So there is never a time when `sq` is cleanly equal to `a*a` or `b*b` (unless `a=b`).

Problem 4: [15 pts] Answer each question below.

(*a*) A company has two teams, A (very good) and C (slackers) working on modules and a testbench for an important product. Describe the following consequences:

☑ The A team works on the modules and the C team works on the testbench. A possible bad outcome is:

The testbench passes the modules, indicating that the modules produce the correct outputs. The company then manufactures the modules and ships them to customers. The customers discover flaws in the modules that the testbench should have, but didn't, uncover. Fortunately the customers weren't avionics or medical equipment manufacturers.

☑ The A team works on the testbench and the C team works on the modules. A possible bad outcome is:

The testbench correctly identifies erroneous module outputs and the C team fixes problem after problem identified by the testbench until eventually they have a set of modules that the testbench passes. The company ships the modules to customers and customers find that the modules do indeed work flawlessly. The only problem is that the modules cost twice as much as competitors' products and use five times the energy. The customers go out of business and so there are no follow-on orders.

Grading Note: Many students assumed that the C team could not get the modules working at all. That would be the F team. We should assume that experienced practicing engineers can get things working and won't be tripped up by problems that might have plagued them in their student days.

(*b*) In typical use when running simulation a testbench generates inputs for a module-under-test and the outputs are checked by the testbench to see whether they are correct. After running synthesis we learn how fast the module is. If simulation is computing the module outputs why can't it tell us how fast the module is?

☑ Synthesis can provide timing information and simulation can't because:

Determining what the output of a module is, is not the same thing as determining when that output will arrive (the delay of the output). To determine timing one needs to know the target technology, and that is not provided to the simulator. Further, one needs to optimize a design, and that's also not something a simulator does. A synthesis program reads the Verilog, a target technology (in the form of a design kit) and transforms the original design into one using components from the target technology, and then optimizes the design to meet a timing constraint at minimum cost.

(*c*) A gadget can be build using an ASIC or an FPGA. Describe which is more appropriate for each situation below.

☑ The gadget must be working within a month. ◯ *ASIC* or ⊗ *FPGA*. ☑ Explain.

An ASIC must be manufactured, a time consuming process that can last months.

☑ Per-gadget cost must be under $1000. Only ten will be made. ◯ *ASIC* or ⊗ *FPGA*. ☑ Explain.

Full-Credit Answer: One can easily buy one FPGA for under $1000, but the minimum order for an ASIC is thousands of units.

Explanation: The minimum number of ASICs that can be manufactured is one wafer with, which might fit hundreds of chips. To make a wafer one must make shadow masks, which themselves aren't cheap. So it makes no sense to use an ASIC target for only ten chips. In contrast, an FPGA is programmed after it is manufactured. Programming an FPGA is more like writing memory. So even if you just buy ten FPFAs there are others buying the same model of FPGA and sharing the development costs.

☑ Per-gadget cost must be under $100. Ten thousand will be made. ⊗ *ASIC* or ◯ *FPGA*. ☑ Explain.

The high costs of setting up an ASIC target can be divided by the 10,000 units that will be sold, resulting in a cost that might be lower than an FPGA.