Name Solution

Digital Design using HDLs
EE 4755
Midterm Examination

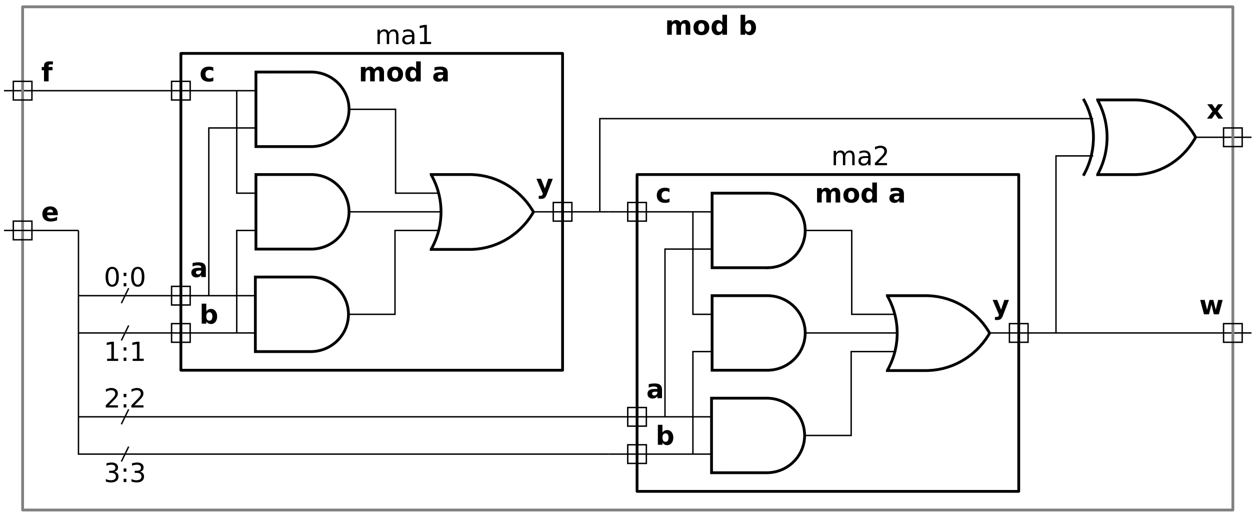
Friday, 21 October 2016 12:30–13:20 CDT

Problem 1		(20 pts)
Problem 2		(20 pts)
Problem 3		(20 pts)
Problem 4		(10 pts)
Problem 5		(10 pts)
Problem 6		(20 pts)
Exam Total		(100 pts)

Alias Loose bits sink chips.

Good Luck!

Problem 1: [20 pts] Write a Verilog description of the hardware illustrated below. The description must include the modules and instantiations as illustrated. The description can be behavioral or structural, but it must be synthesizable.



☑ Verilog corresponding to illustrated hardware.

☑ Show instantiations, ☑ Verilog for instantiated module(s), ☑ and all module ports.

Solution appears below.

Grading Note: Many students chose to provide an explicit structural description, which is the most tedious descriptive style. In an explicit structural description `moda` *uses four primitive instantiations plus a declaration for three wires. As can be seen from the solution the implicit structural description is just one line.*

In many solutions for `modb` *the output of* `ma2` *was connected to an intermediate uwire, and an* `assign` *statement was used to connect the uwire to the module output. As can be seen from the solution, the* `ma2` *output can connect directly to the* `modb` *output.*

```
// SOLUTION
module moda(output uwire y, input uwire c, a, b);
   assign y = a && c || a && b || b && c;
endmodule

module modb(output uwire x, w, input uwire [3:0] e, input uwire f);

   uwire y1;

   moda ma1(y1,f,e[0],e[1]);
   moda ma2(w,y1,e[2],e[3]);

   assign x = y1 ^ w;

endmodule
```

Problem 2: [20 pts] Appearing below is the `lookup_elt` module from Homework 4 and following that an incomplete module named `match_amt_elt`. Complete `match_amt_elt` so that the value at output port `md` is set to the number of bits in `clook` that match corresponding bits in `celt`. For example, if `clook=5'b00111` and `celt=5'b00111` then `md` should be 5, if `clook=5'b00101` and `celt=5'b00111` then `md` should be 4, and if `clook=5'b11000` and `celt=5'b00111` then `md` should be 0. Code must be synthesizable, but can be behavioral or structural.

☑ Complete the module so that `md` is set to the number of matching bits.

☑ Make sure that `md` is declared with sufficient width.

```
module lookup_elt #( int charsz = 32 )  // This module is for reference only.
   ( output logic match, input uwire [charsz-1:0] char_lookup, char_elt );
   always_comb match = char_lookup == char_elt;
endmodule
```

The solution appears below.

For the size of `md`, notice that `md` must represent `charsz+1` distinct values, 0 to `charsz`. Therefore `clog2(charsz+1)` bits are needed. *Grading note: Full credit was given for almost any declaration that contained* `clog2(charsz)`*, not just those which were perfectly correct. Points were deducted for constant answers such as* `[5:0]` *since they only work for the default value of* `charsz`*.*

To count the number of matching bits a loop is used to iterate over the bits and a simple comparison is used to find matches.

Grading Notes: There was no reason to use `lookup_elt`*, it was put in the problem only to help people get started. A correct solution could use* `lookup_elt`*, however it had to be instantiated with a* `charsz=1`*.*

In too many solutions there was confusion between procedural code (code starting with some kind of `always`*) and structural code (module declarations and* `assign` *statements).*

```
module match_amt_elt
  #( int charsz = 32 )
   ( output logic     [$clog2(charsz+1)-1:0]     md,   // SOLUTION (The [$clog..])
     input uwire [charsz-1:0] clook,
     input uwire [charsz-1:0] celt);

   // SOLUTION
   always_comb begin
      md = 0;
      for ( int i=0; i<charsz; i++ ) if ( clook[i] == celt[i] ) md++;
   end

endmodule
```

Problem 3: [20 pts] Show the hardware that will be synthesized for the modules below.

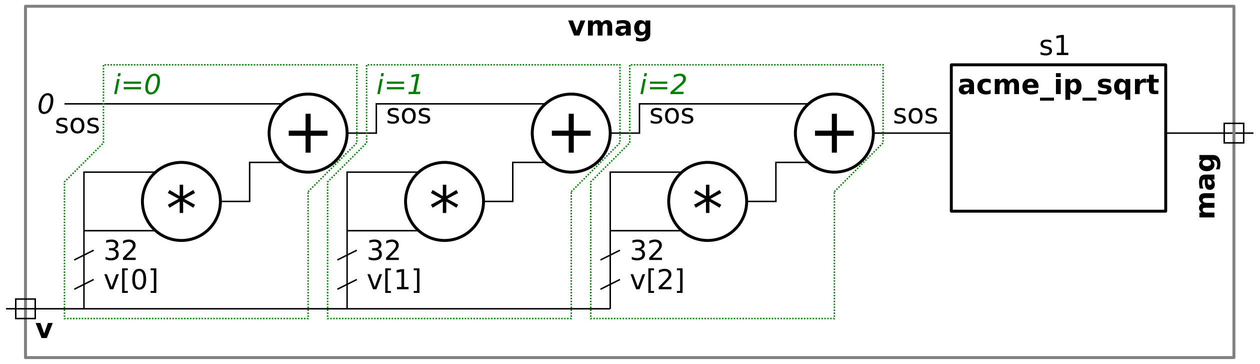
(*a*) Show the hardware that will be inferred for the module below. Show `acme_ip_sqrt` as a box.

```
module vmag( output uwire [31:0] mag,  input uwire signed [31:0] v [3] );

   logic [63:0] sos;
   acme_ip_sqrt #(32) s1(mag,sos);

   always_comb begin
      sos = 0;
      for ( int i=0; i<3; i++ ) sos += v[i] * v[i];
   end

endmodule
```

☑ Show inferred hardware. ☑ Don't forget `acme_ip_sqrt`.

☑ Clearly show input and output ports of `vmag`.

Solution appears below.



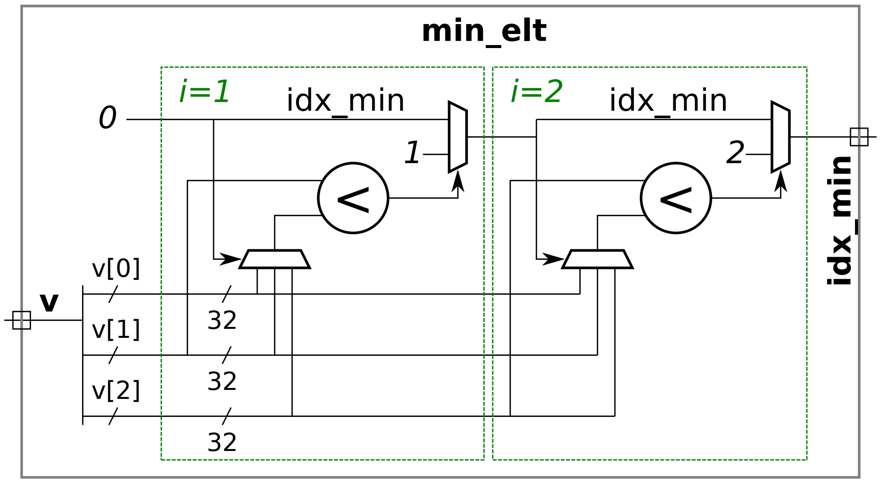
Problem 3, continued:

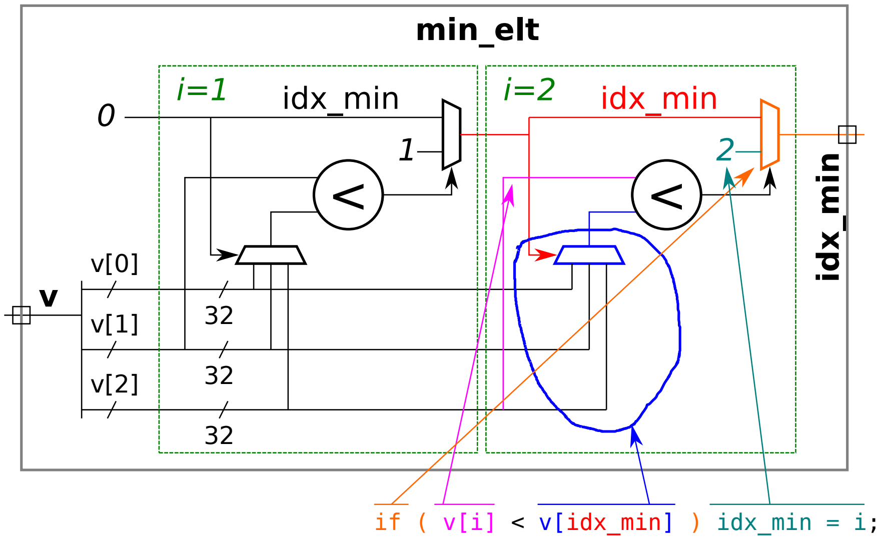
(*b*) Show the hardware that will be inferred for the module below, before and after optimization. *Note: In the original exam the input was named* `vi`*.*

```
module min_elt( output logic [1:0] idx_min, input uwire signed [31:0] v [3] );
   always_comb begin
      idx_min = 0;
      for ( int i=1; i<3; i++ ) if ( v[i] < v[idx_min] ) idx_min = i;
   end
endmodule
```

☑ Show inferred hardware. ☑ Clearly show input and output ports.

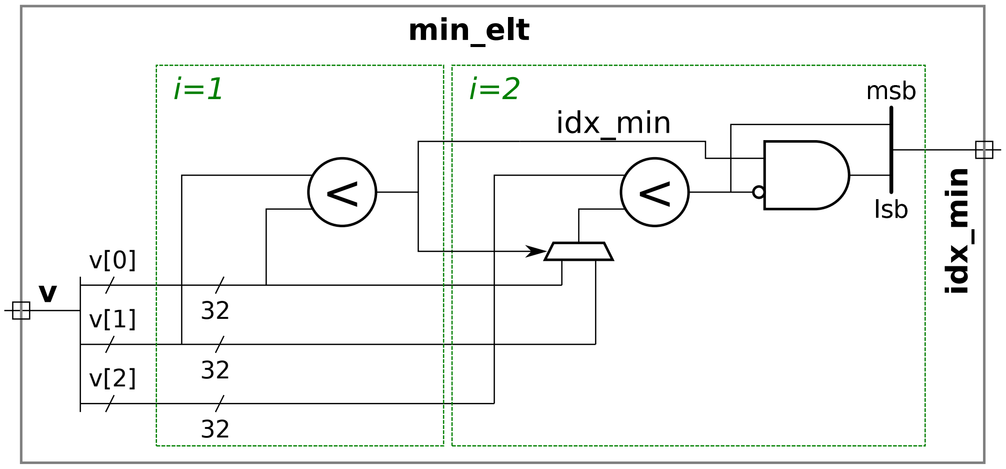
Solution appears below in a plain form, followed by a version in which the hardware corresponding to the different parts of the `if` statement is highlighted. *Grading Note: A common difficulty was coming up with the hardware for* `v[idx_min]`*.*





☑ Show hardware after some optimization.

Solution appears below. The 3-input mux at `i=1` has been eliminated because it always selected element 0. The 3-input mux at `i=2` was replaced by a 2-input mux because the select input would never be 2. The 2-input mux at `i=1` was eliminated since the select signal has the same value as the output. The 2-input mux at `i=2` was replaced by uwire and a single AND gate (with a bubbled input).

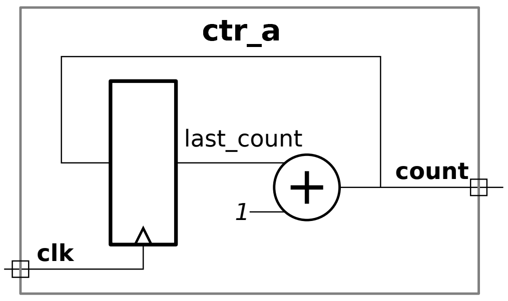


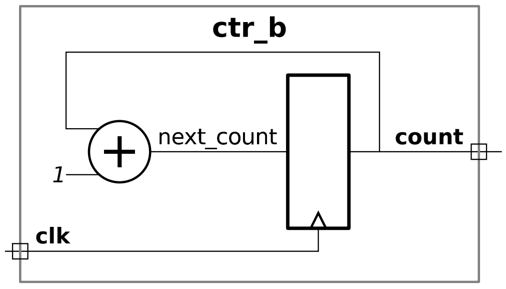
Problem 4: [10 pts] Appearing in this problem are several variations on a counter.

(*a*) Show the hardware inferred for each counter below.

```
module ctr_a( output uwire [9:0] count, input clk );

   logic [9:0] last_count;
   assign count = last_count + 1;
   always_ff @( posedge clk ) last_count <= count;

endmodule

module ctr_b( output logic [9:0] count, input clk );

   uwire [9:0] next_count = count + 1;
   always_ff @( posedge clk ) count <= next_count;

endmodule
```

☑ Inferred hardware for ☑ ctr_a and ☑ ctr_b.

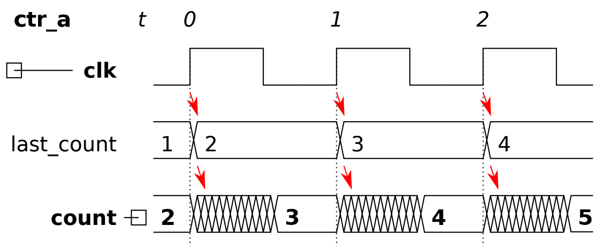


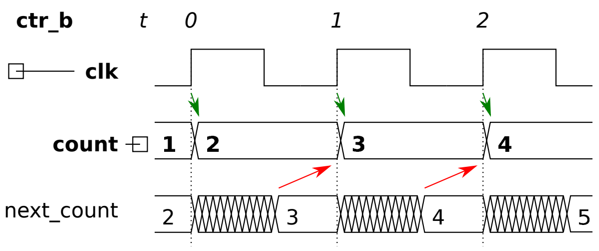


(*b*) There is a big difference in the timing of the outputs of ctr_a and ctr_b. Explain the difference and illustrate with a timing diagram.

☑ Difference between two modules. ☑ Timing Diagram.

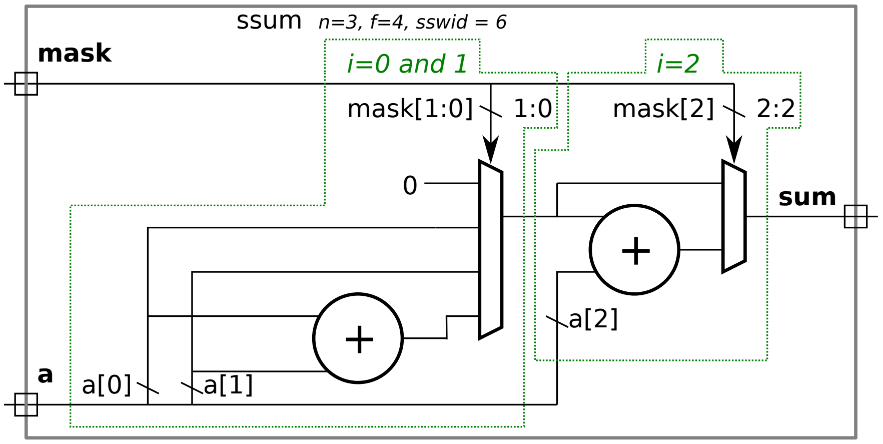
In `ctr_a` the module output, `count`, is connected to the output of an adder. That means the value at the output will not be stable until later in the clock cycle. See the left-side timing diagram below. External hardware could not do anything with the value other than clocking it into a register for use in the next clock cycle. In contrast, the `ctr_b` module output, `count`, is connected to a register output, and so it is available for use at the beginning of the clock cycle.





Problem 5: [10 pts] Appearing below is the solution to the 2015 midterm exam Problem 2. Estimate the cost of this module as illustrated but use variable s for the number of bits in `sum` (shown as `sswid`) and in each `a` element (shown as parameter `f`). Assume that the cost of a BFA is 10 units and that the cost of a n-input AND and OR gate is $n-1$ units. Take into account the 0 input to one of the multiplexors.

```
module ssum #( int n = 3,      int f = 4,     int swid = f + $clog2(n) )
             ( output logic [swid-1:0] sum,
               input uwire [n-1:0] mask,      input uwire [f-1:0] a[n] );
   always @* begin
      sum = 0;
      for ( int i=0; i<n; i++ ) if ( mask[i] ) sum += a[i];
   end
endmodule
```



☑ Cost of illustrated hardware. ☑ Account for 0 mux input.

There are two adders, each uses s bits. Since the cost of a BFA is 10 units, the cost of the two adders is $2 \times 10s = 20s$ units.

A two-input mux uses three 2-input gates per bit, so the total cost of the second mux is $3s$ units. In general, an n-input, width w mux uses nw 2-input AND gates for selection, n $\lceil \lg n \rceil$-input AND gates for decoding, and w n-input OR gates. Without optimization, the 4-input mux would cost $4s + 4 + 3s = 7s + 4$ units. The total cost without optimization is $20s + 7s + 4 + 3s = 30s + 4$ units.

Because one of the inputs to the 4-input mux is zero all of the logic connecting to that input can be eliminated, and the OR gates can be reduced from 4 to 3 inputs. So the optimized cost of the 4-input mux is $3s + 3 + 2s = 5s + 3$ units. The total cost with optimization is $\boxed{20s + 5s + 3 + 3s = 28s + 3 \text{ units}}$.

Grading Note: Way too many students did not multiply the cost of a BFA by the number of bits in the quantities being added.

Problem 6: [20 pts] Answer each question below.

(*a*) Show the values of the variables as indicated below:

Solution appears below. Notice that the difference between `x1` and `x2` is with the bit numbering. In the `e[0]+'hf` assignment the `'hf` (15 represented as a hexadecimal digit) is being added to the least-significant 4-bits of `e`. The result of that addition is $4_{16} + f_{16} = 13_{16}$, and it is placed in the 4 least significant bits of `e` without modifying the other bits of `e`. The assignment to `e[0][0]` is similar, except that it operates only on the least-significant bit of `e`.

```
module tryout();
   logic [15:0] a;
   logic [0:15] b;
   logic [3:0][3:0] e;
   logic [3:0]  x1, x2;

   initial begin

      a = 16'h1234;
      x1 = a[3:0];                  // ☑  Value of x1 is: 4

      b = 16'h1234;
      x2 = b[0:3];                  // ☑  Value of x2 is: 1

      e = 16'h1234;
      e[0] = e[0] + 'hf;            // ☑  Value of e is:  16'h1233

      e = 16'h1234;
      e[0][0] = e[0][0] + 'hf;      // ☑  Value of e is:  16'h1235

   end
endmodule
```

(*b*) Describe something that can be done during elaboration that cannot be done during simulation, and something that can be done during simulation, that cannot be done during elaboration.

☑ Something that can be done during elaboration but **not during simulation** is:

During elaboration one can use a generate loop to instantiate modules.

☑ Something that can be done during simulation but **not during elaboration** is:

During simulation one can compute values that depend on module inputs.

(*c*) Appearing below are two alternatives for an integer division module, *Plan A* and *Plan B*. Both are impractical, but Plan A is not even synthesizable.

```
module div_plan_a #( int w = 16 ) ( output logic [w-1:0] quo, input uwire [w-1:0] a, b );
   always_comb begin
      for ( quo = 0;  a > quo * b;  quo++ );
   end
endmodule
```

```
module div_plan_b #( int w = 16 ) ( output logic [w-1:0] quo, input uwire [w-1:0] a, b );
   localparam int LIMIT = 1 << w;
   always_comb begin
      quo = 0;
      for ( int i=0; i<LIMIT; i++ ) if ( a < i * b ) quo++;
   end
endmodule
```

✓ Why isn't Plan A synthesizable? Be specific as possible.

It is not synthesizable because the number of iterations in the loop can not be determined at elaboration time.

✓ What might be a practical objection to the Plan B approach?

Because 2^w multipliers and multiplexors are used the cost is ridiculously high for even moderate values of `w`. For example, for the default value of `16`, there would be `65536` multipliers and muxen. Even if the synthesis program simplified this to `65536` adders, the cost would still be enormous.

(*d*) The `magfp` module below is not synthesizable due to the use of the `real` data type. How would the module need to be changed so that it would be synthesizable and would operate on floating-point values.

```
module magfp( output real mag, input real vi [3] );
   real sos;
   sqrt #(32) s1(mag,sos);
   always_comb begin
      sos = 0;
      for ( int i=0; i<3; i++ ) sos += vi[i] * vi[i];
   end
endmodule
```

✓ Show changes to port declaration for synthesizability.

Change `real` to `[63:0]`.

✓ Explain with a few examples how the rest of the code would need to be changed.

The arithmetic operations would have to be replaced by bit-level operations to perform the floating-point arithmetic. This might be done by instantiating FP multipliers and adders from an IP library (such as ChipWare) or by writing the Verilog to implement FP arithmetic yourself.