Name Solution

# Digital Design using HDLs

# EE 4755

# Midterm Examination

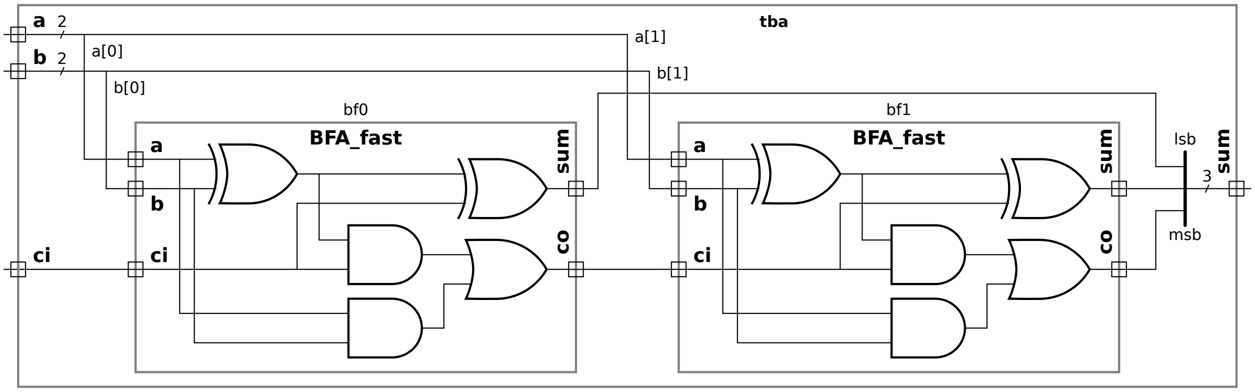
Monday, 16 October 2017 9:30–10:20 CDT

| | | |
|---|---|---|
| Problem 1 | | (20 pts) |
| Problem 2 | | (20 pts) |
| Problem 3 | | (20 pts) |
| Problem 4 | | (15 pts) |
| Problem 5 | | (10 pts) |
| Problem 6 | | (15 pts) |
| Exam Total | | (100 pts) |

Alias Even Ireland.

*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
module BFA_fast( output uwire sum, co, input uwire a, b, ci );

   // Note: axb explicitly computed once and used twice.
   uwire  axb = a ^ b;
   assign sum = axb ^ ci;
   assign co = axb && ci || a && b;

endmodule

module tba( output uwire [2:0] sum,   input uwire [1:0] a, b,   input uwire ci );

   uwire  c;
   BFA_fast bf0( sum[0], c,      a[0], b[0], ci );
   BFA_fast bf1( sum[1], sum[2], a[1], b[1], c  );

endmodule
```

Problem 2: [20 pts] Appearing below is a partially completed recursive description of an $n = 2^b$-input, $w$-bit multiplexor, which is a generalized version of the multiplexors appearing in Homework 1. Complete it.

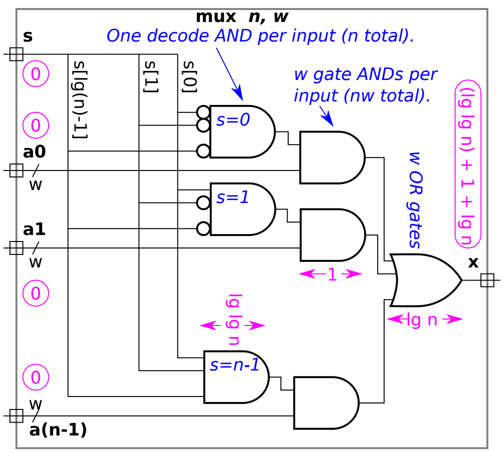
☑ Fill in the condition and code for the terminating case.

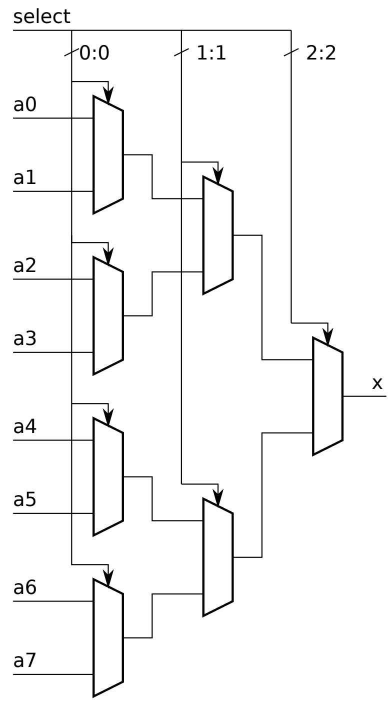
☑ Complete recursive case, including the instantiation port and parameter connections (look for FILL IN).

```
module muxn #( int w = 5, int b = 4, int n = 1 << b )
   ( output uwire [w-1:0] x,  input uwire [b-1:0] sel, input uwire [w-1:0] a[0:n-1] );

   if (  b == 1     )  // Terminating Case Condition              <----  ☑  FILL IN
     begin
        // Terminating Case

        assign x = a[sel];




     end else begin
        // Recursive Case

        uwire [w-1:0] y[2];

        // Instantiate two n/2-input muxen, and connect each to half the inputs.
        //
        //         ----      ----                                <----  ☑  FILL IN
        muxn #( .w(  W ), .b( b-1 ) ) mlo( y[0],  sel[b-2:0],  a[   0 : n/2-1 ] );


        //         ----      ----                      -----     <----  ☑  FILL IN
        muxn #( .w(  W ), .b( b-1 ) ) mhi( y[1],  sel[ b-2:0 ],  a[ n/2 : n-1   ] );




        // Instantiate one 2-input mux.
        //
        //         ----      ----       -------------------      <----  ☑  FILL IN
        muxn #( .w(  W ), .b( 1 ) ) m2(   x,   sel[b-1],  y    );


   end

endmodule
```

Problem 3: [20 pts] Appearing below to the right is an 8-input multiplexor constructed from 2-input multiplexors using the technique from Homework 1 and from the previous problem. Call a multiplexor constructed this way a *tree mux*. Appearing below to the left is a diagram showing a *flat mux*, the kind usually used in class. The flat mux diagram shows a timing analysis based on the simple model, and some details about cost.

For reference: $\sum_{i=0}^{b-1} a2^i = a(2^b - 1)$. Assume that $n$ is a power of 2.




(*a*) Compute the cost of an $n$-input, $w$-bit flat mux using the simple model and without optimization.

☑ Cost of flat mux **in terms of** $n$ **and** $w$.

As can be seen from the diagram, the $n$ decode gates each have $\lg n$ inputs, for a total cost of $n(\lg n - 1)$. The gate AND gates each have two inputs and there are $nw$ of them, for a total cost of $nw$ units. The OR gate has $n$ inputs and there are $w$ of them, so their cost is $(n-1)w$ units. The total cost is then $n(\lg n - 1) + 2nw - w$ units.

(*b*) Compute the cost of an $n$-input, $w$-bit tree mux using the simple model.

☑ Cost of tree mux **in terms of** $n$ **and** $w$. ☑ Describe assumptions made about 2-input mux implementation.

As can be seen in the diagram, in the first column there are $n/2 = 2^{b-1}$ multiplexors, where $n = 2^b$. The second column has $2^{b-2}$ multiplexors, and so on, the last column has $2^0 = 1$ multiplexor. The total number of multiplexors is $\sum_{i=0}^{b-1} 2^i = 2^b - 1$ multiplexors. The cost of a 2-input, $w$-bit mux flat is $3w$ units (see the previous part) and so the total cost of the tree mux is $3w(2^b - 1) = 3w(n - 1)$.

(*c*) Compute the delay of an $n$-input, $w$-bit tree mux using the simple model.

☑ Delay of tree mux **in terms of** $n$ **and** $w$.

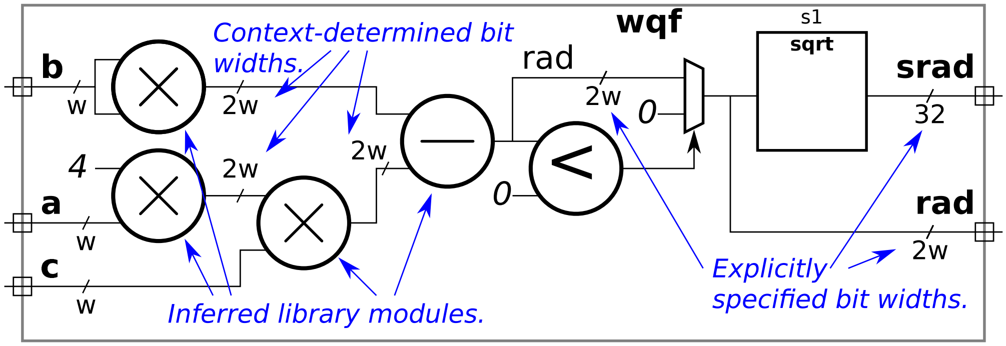
The critical path passes through $\lg n$ layers (columns in the diagram). Each layer is a 2-input mux, in which the critical path passes through an AND gate and a OR gate, each of two inputs, so the delay is 2 units per layer. Therefore the delay is $2 \lg n$ units.

Problem 4: [15 pts] Show the hardware that will be synthesized for the modules below.

($a$) Show the hardware that will be inferred for the module below, including the minimum number of bits in each wire. Assume that `sqrt` is defined in a library somewhere.

```
module wqf
  #( int w = 16 )
   ( output logic signed [2*w-1:0] rad,
     output uwire [31:0] srad,
     input uwire [w-1:0] a, b, c );

   sqrt #(32,2*w) s1(srad,rad);

   always_comb begin

      rad = b*b - 4 * a * c;
      if ( rad < 0 ) rad = 0;

   end

endmodule
```

☑ Show inferred hardware. ☑ Show minimum correct bit widths.

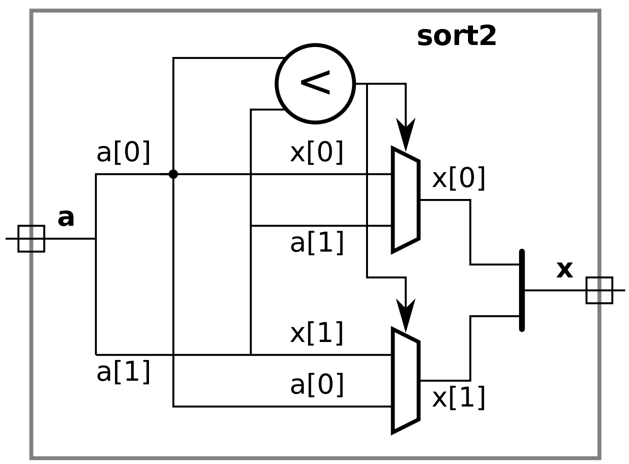



Solution appears above. Note that the basic arithmetic operators are replaced by library modules (shown as circles) provided by the synthesis program, whereas the `sqrt` module is explicitly instantiated in the module above. The multiplexor is inferred from the `if` statement. The select signal is connected to a comparison module, however that could easily be optimized into a connection to the sign bit of output of the subtractor. Similarly the $\times 4$ multiplier could have been optimized to a bit renumbering. But the question asks for *inferred* hardware, and so even these easy optimizations are omitted. The sizes of the wires connected to module ports are given explicitly in the `wqf` module, whereas widths of the internal wires are determined using Verilog rules for bit widths. Under those rules multiplication and subtraction arguments' bit widths are context-determined. Note that `rad` is explicitly sized to $2w$ bits, this context at the subtract output determines the size as the subtract inputs, which in turn determines the width needed for the multiplies.

(*b*) Show the hardware that will be inferred for the module below.

```
module sort2 #( int w = 4 )
   ( output logic [w-1:0] x[2], input uwire [w-1:0] a[2] );

   always_comb begin

      for ( int i=0; i<2; i++ ) x[i] = a[i];
      if ( a[0] < a[1] ) begin x[0] = a[1]; x[1] = a[0]; end

   end

endmodule
```

☑ Show inferred hardware.




Solution appears above. Note that the effect of the `for` loop is only to make `x[0]` another name for `a[0]` and `x[1]` another name for `a[1]`.

Problem 5: [10 pts] Answer each question below.

(*a*) The `mux2` module below uses implicit structural code. Modify it so that it uses behavioral (procedural) code.

```
module mux2 #( int w = 16 )
   ( output uwire [w-1:0] x,
     input uwire s, input uwire [w-1:0] a,b );

   assign  x = s == 0 ? a : b;

endmodule


// SOLUTION
module mux2 #( int w = 16 )
   ( output logic [w-1:0] x,
     input uwire s, input uwire [w-1:0] a,b );

   always_comb x = s == 0 ? a : b;

endmodule
```

☑ Modify so that is procedural. ☑ Change ports if necessary.

Solution appears above. Note that in addition to changing **assign** to **always_comb**, the kind of object of the input port was changed from net to var. (**uwire** is an object of kind net with a default data type of **logic**, and **logic** is a data type with a default object kind of var.)

(*b*) Modify the module port and parameter declarations below so that the Verilog is correct. Do not modify the contents of the module itself. Note that `opt` is not defined, but that it should be. *Note: In the original exam* `assign` *was omitted from the module body, making the problem impossible to solve.*

```
module sum_or_dff
  #( int w = 16 )
   ( output uwire [w-1:0] x,
     input uwire [w-1:0] a, b );

   if ( opt == 0 ) assign x = a+b; else assign x = a-b;

endmodule
```

```
module sum_or_dff
  #( int w = 16, int opt = 1  )
   ( output uwire [w-1:0] x,
     input uwire [w-1:0] a, b );

   if ( opt == 0 ) assign x = a+b; else assign x = a-b;

endmodule
```

☑ Modify port and parameter declarations for correctness.

Solution appears above. The `if` statement, because it is in module scope, is a generate statement and therefore the condition must be an elaboration-time constant. For that reason `opt` is made a parameter.

Problem 6: [15 pts] Answer each question below.

(*a*) Why is `always_comb` preferred over `always @( x or y or .. )` when describing combinational logic?

☑ `always_comb` preferred because ...

... there is no need to take the trouble to list all of the live-in objects nor is there the risk of omitting one.

☑ What is the risk with `always @( x or y or ..  )`?

If a live-in object is omitted from the sensitivity list, code in the block will not be re-executed when the value of the omitted object changes but other variables don't change. For example, consider the `sum` module below. The intent is hardware that adds three numbers together. But because `z` was omitted the value of output `a` will not be "correct" if `z` changes but `x` and `y` stay the same. In general, the simulation might not produce the answers that are expected and the synthesis program will infer a latch (or latches) rather than combinational logic.

```
// Module illustrating error easily made using old-school Verilog sensitivity lists.
module sum(output logic [15:0] a, input uwire [15:0] x, y, z );
 always @( x or y ) a = x + y + z;
endmodule
```

(*b*) Describe what the technology mapping step of synthesis is, and the kind of optimizations that need to be performed after technology mapping.

☑ Technology mapping is:

the substitution of generic components in the inferred hardware with components in the target technology being synthesized. For example, a three-input AND gate (a generic component) might be replaced by `ASx9AND4`, a four-input AND gate in Acme Silicon's x9 ASIC cell library. (Acme Silicon's x9 ASIC cell library does not have a three-input AND gate.) Note: Acme Silicon is a fictional silicon foundry made up for this problem's solution.

☑ Optimizations that must be performed after technology mapping:

Most cost reduction optimizations must be done after technology mapping because only after technology mapping are the cost and timing of components known.

(*c*) The module below adds a real and an integer and assigns the sum (in real format) to its output. It is valid Verilog but is not synthesizable by Owr EDA software. So, you call Owr EDA and ask, "why not?". They answer, "because it is impossible to add an integer to a real." Is that the real reason? Explain.

```
module plusri (output real sum, input real a, input [20:0] x);
   assign       sum = a + x;
endmodule
```

☑ Reason a+x not synthesizable by Owr EDA software:

If Owr EDA wanted to they could infer an integer-to-real conversion module to convert `x` to a real and a real addition module to compute the sum. There are no fundamental reasons why a synthesis program can not have such features. They did not do so because it never made it to the top of their to do list, perhaps.