Name Solution

Digital Design Using HDLs
LSU EE 4755
Midterm Examination

Wednesday, 30 October 2019 10:30–11:20 CDT

Problem 1		(20 pts)
Problem 2		(25 pts)
Problem 3		(27 pts)
Problem 4		(28 pts)
Exam Total		(100 pts)

Alias That's ··· all.

Good Luck!

Problem 1: [20 pts] Appearing below is one of the solutions to Homework 2, the count leading zeros module.

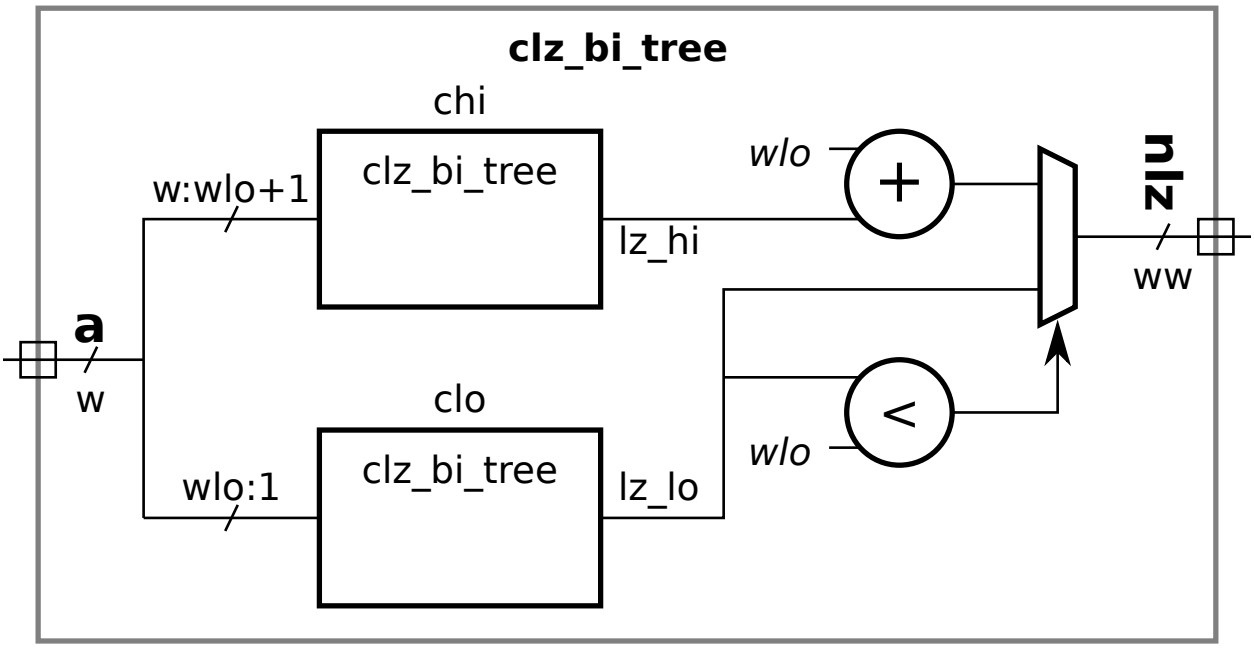
```
module clz_bi_tree #( int w = 19, int ww = $clog2(w+1) )
                    ( output uwire [ww:1] nlz,   input uwire [w:1] a );
   if ( w == 1 ) begin
      assign nlz = ~ a;
   end else begin
      localparam int wlo = w/2,             whi = w - wlo;
      localparam int wwlo = $clog2(wlo+1),  wwhi = $clog2(whi+1);
      uwire [wwlo:1] lz_lo;
      uwire [wwhi:1] lz_hi;
      clz_bi_tree #(wlo) clo( lz_lo, a[wlo:1]   );
      clz_bi_tree #(whi) chi( lz_hi, a[w:wlo+1] );
      assign nlz = lz_lo < wlo ? lz_lo : wlo + lz_hi;
   end
endmodule
```

Show the hardware that will be inferred for the module for $w > 1$. Just show one level, don't show what is inside of `clo` and `chi`.

☑ Show synthesized hardware for one level. ☑ Be sure to show `clo` and `chi` (but not their contents).

☑ Clearly show module input and output ports, ☑ and show bit range in connections.

The solution appears below. Because $w > 1$ the terminal case is not elaborated and so not inferred. Of course, there is no hardware for computing elaboration-time constants such as `wwlo`.



Problem 2: [25 pts] In Homework 2 a clz (count leading zeros) module was constructed recursively by splitting the input bit vector and connecting each half to a smaller instance. The incomplete module below is similar except that the input vector is to be split into thirds and each third connected to a recursive instance. Complete the module.

☑ Complete so that `clz_tri_tree` computes clz.

The solution appears below.

```
module clz_tri_tree
  #( int w = 19, int ww = $clog2(w+1) )
   ( output uwire [ww-1:0] nlz, input uwire [w-1:0] a );

   if ( w == 1 ) begin

      assign nlz = ~ a;

      // SOLUTION: Add a case for w=2 to avoid a zero-bit recursive instance.
   end else if ( w == 2 ) begin

      assign nlz = a[0] ? 0 : a[1] ? 1 : 2;

   end else begin

      // SOLUTION: Divide bits between modules, be sure not to loose any.
      localparam int wlo = w/3;
      localparam int wmi = wlo;
      localparam int whi = w - wlo - wmi;

      localparam int wwlo = $clog2(wlo+1), wwmi = $clog2(wmi+1), wwhi = $clog2(whi+1);
      uwire [wwlo-1:0] lz_lo;
      uwire [wwmi-1:0] lz_mi;
      uwire [wwhi-1:0] lz_hi;

      // SOLUTION: Divide a between modules.
      clz_tri_tree #(wlo) clo( lz_lo, a[ wlo-1   : 0     ] );
      clz_tri_tree #(wmi) cmi( lz_mi, a[ w-whi-1 : wlo   ] );
      clz_tri_tree #(whi) chi( lz_hi, a[ w-1     : w-whi ] );

      // SOLUTION: Combine the results of the three modules.
      assign nlz = lz_lo < wlo  ?  lz_lo        :
                   lz_mi < wmi  ?  wlo + lz_mi  :  wlo + wmi + lz_hi;

   end

endmodule
```

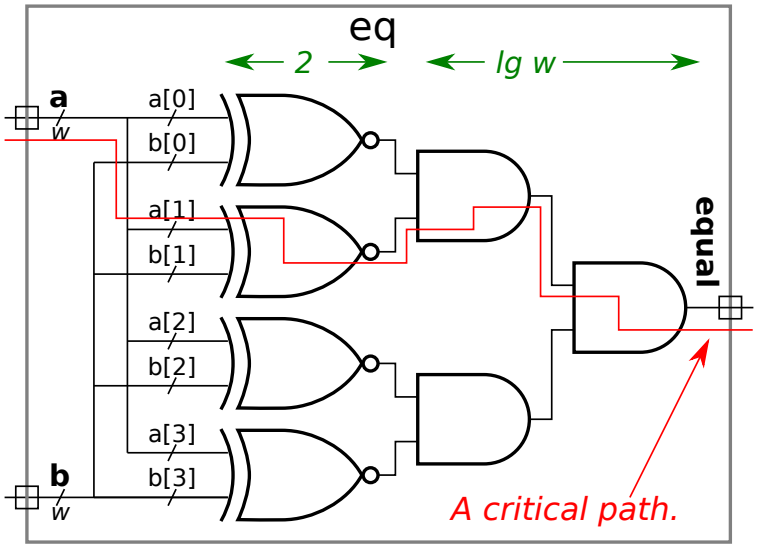
Problem 3: [27 pts] Appearing below are modules that test if two bit vectors are equal in some way.

(*a*) Show the hardware for the module below at the default size using basic gates: AND, OR, XOR, NOTs, and bubbled inputs and outputs. **Do not** use something like `==`.

```
module eq #( int w = 4 )( output uwire equal, input uwire [w-1:0] a, b );
   assign equal = a == b;
endmodule
```

☑ Show hardware using basic gates at default size.

The solution appears below with some colored labels to help with the next subproblem. Note that—never forget that—equality is tested using XNOR (exclusive nor) gates.



(*b*) Show the cost and delay of the module in terms of w (the value of parameter `w`) using the simple model.

☑ In terms of w: ☑ Cost and ☑ Delay.

The cost is $[3w + w - 1]\,\mathrm{u_c} = [4w - 1]\,\mathrm{u_c}$. The $3w$ term is for the XNOR gates and $w - 1$ term is for the big AND gate. (In the solution above three 2-input AND gates are shown rather than one 4-input AND gate.) The delay is $[2 + \lceil \lg w \rceil]\ \mathrm{u_t}$, the 2 term is for an XNOR gate and the $\lg w$ term is for a path through the big AND gate.

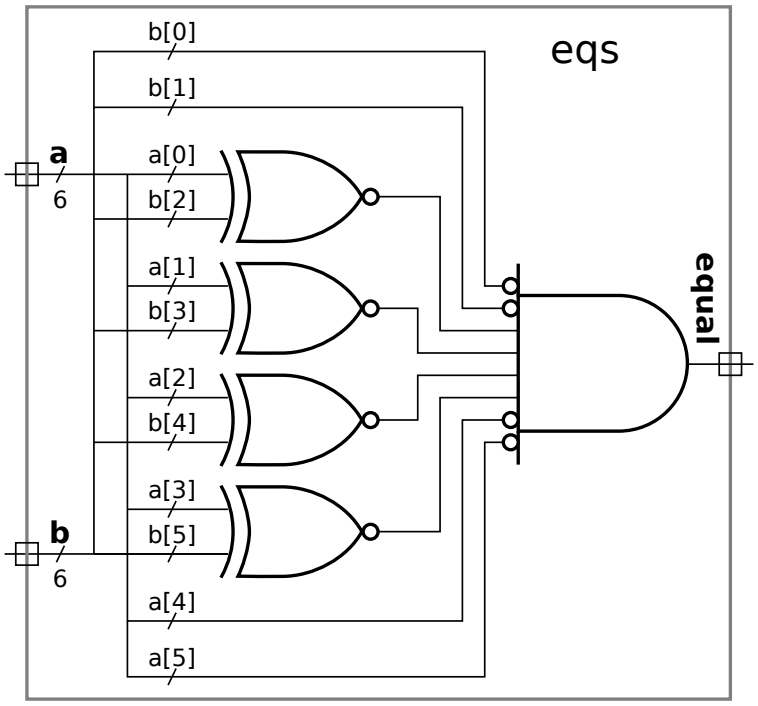
To compute delay a critical path is needed. A critical path for the equality unit is shown above in red, starting at `a[1]`. Because of symmetry in the equality unit the critical path could have started at any input bit. The path through an XNOR is two gates, and a path through the big AND is $\lceil \lg w \rceil$ gates.

(*c*) The module below also tests equality but it does so after shifting the first operand. Show the hardware in terms of basic gates after optimization.

```
module eqs #( int w = 6, int s = 2 )( output uwire equal, input uwire [w-1:0] a, b );
   localparam logic [w+s-1:0] zero = 0;
   assign equal = zero + ( a << s ) == b;
endmodule
```

☑ Show hardware at default size after optimization.

The solution appears below. Because the shift is by a constant amount no shifter is needed, instead the bit positions are adjusted (which is why, for example, `b[2]` is compared to `a[0]`). Because we are adding zero no adder is needed. Because of the shift the low bits of `b` and the high bits of `a` are compared to zero.



(*d*) The module below performs a different operation than the one above. Explain the difference and show an example.

```
module eqt #( int w = 6, int s = 2 ) ( output uwire equal, input uwire [w-1:0] a, b );

   assign equal = ( a << s ) == b;

endmodule
```

☑ Difference between operation `eqs` and `eqt`.

☑ Show a value for `a` and `b` for which the output of `eqs` and `eqt` are different.

In module `eqs` the s MSB are compared to zero, whereas in `eqt` the s MSB are ignored. For example, consider $w = 6$ and $s = 2$, and for $a = 10\,1111_2$ and $b = 11\,1100_2$. Module `eqs` finds them not equal (because eight-bit quantities $1011\,1100_2 \neq 0011\,1100_2$) but `eqt` finds them equal (because six-bit quantities $11\,1100_2 = 11\,1100_2$).

Problem 4: [28 pts] Answer each question below.

(*a*) Appearing below is synthesis data taken from the solution to Homework 2. The `Delay Target` column shows the maximum delay constraint given to the synthesis program.

Module Name	Area	Delay Actual	Delay Target
clz_w32	26290	3.110	10.000 ns
clz_tree_w32	21706	1.425	10.000 ns
clz_w32_1	36476	1.007	0.100 ns
clz_tree_w32_5	37356	0.577	0.100 ns

☑ In general, which result should be used if the only goal were to minimize area,
the results for the ☑ 10.0 ns *Target* or for the ◯ 0.1 ns *Target* ? ☑ Explain.

When the delay target is large the synthesis program is freer to minimize area (cost). It can try different cost-reducing optimizations without having them being rejected because they result in higher delay (as long as that delay is below the delay target).

☑ In general, which result should be used if the only goal were to minimize delay,
the results for the ◯ 10.0 ns *Target* or for the ☑ 0.1 ns *Target* ? ☑ Explain.

The synthesis program first tries to meet the delay target, then reduces cost. If the delay target is very low it will devote all of its effort to reducing delay.

(*b*) Provide w-bit declarations requested below.

```
uwire [ 0 : w-1 ] bit_zero_is_msb;        // SOLUTION
uwire [ w-1 : 0 ] bit_zero_is_lsb;        // SOLUTION
uwire [ w/2 : -w/2 ] bit_zero_is_middle;  // SOLUTION.
```

(*c*) The module fragment below starts with six declarations (the object names starting with `r`), each providing a value (either `a+b` or `x+y`). Some of those declarations will result in compile errors. Identify them and explain the problem. If possible fix the problem without changing the object kind (localparam, uwire, var).

```
module my_mod
  #( int w = 10, int x = 11, int y = 12 )
   ( input uwire [w:1] a, b );

   localparam logic [w:1] r1p = a + b; // SOL: Can't fix, a + b not constant.

   localparam logic [w:1] r2p = x + y; // SOL: Okay.

   uwire [w:1] r1w = a + b;  // SOL: Okay.

   uwire [w:1] r2w = x + y;  // SOL: Okay.

   logic [w:1] r1l = a + b;  // SOL: Wrong, can't continuously assign var type.

   logic [w:1] r2l = x + y;  // SOL: Wrong, can't continuously assign var type.

   // SOLUTION: Fixes:
   logic [w:1] r1l, r2l;
   always_comb begin r1l = a + b;  r2l = x + y;  end

   // The following is not wrong, but it's longer than the original.
   uwire [w:1] r12, r2w;
   assign r1w = a + b;
   assign r2w = x + y;
```

☑ Indicate which ones are wrong and ☑ the reason that they are wrong.

☑ Indicate which can't be fixed and ☑ and explain why not.

The value assigned to a `localparam` must be an elaboration-time constant. That's true for `x+y` because they are parameters, but it's not true for `a+b` because `a` and `b` are module inputs and so could never be elaboration-time constants.

The assignments to `r1w` and `r2w` are fine. SystemVerilog allows a net (including `uwire`) declaration to include a continuous assignment.

The assignments to `r1l` and `r2l` are wrong because var objects can only be assigned in procedural code. That's easy to fix by providing an `always` block, which is shown above.

(Note that a declaration like `logic [w:1] v;` is shorthand for `var logic [w:1] v;` and a declaration like `uwire [w:1] u;` is shorthand for `uwire logic [w:1] u;`.)

Other than for `r1p` the size, type, and kind of `a`, `b`, `x`, and `y` are not a problem. The sum `x+y` is a 32-bit 2-state integer. It's not an error to assign that to a `w`-bit four state type. Also note that the data type for all of the `r[12][pwl]` objects are logic. (Note that `r[12][pwl]` is an ad-hoc regular expression matching the objects being assigned above. Regular expressions are something you should know in general, but not for this course.)

Grading Note: Students had more difficulty with this problem than I expected. As I pointed out in class, if you don't understand the different object kinds (net, var, param) and how they should be used you'll waste lots of time blindly changing things until the error messages go away.

(*d*) Explain what `$realtobits` does, and what hardware will be synthesized for it, if any.

```
always_comb begin

   x = $realtobits(r);

end
```

☑ Purpose of `realtobits`.

The `realbits` system task is used to move a set of bits from an object declared `real` to one declared as some kind of integer (say, `logic [63:0]`). The bits are moved unchanged. If, instead the assignment were `x=r;` the simulator would convert the `real` value in `r` to an integer.

☑ Synthesized hardware.

None. If we were to draw a diagram, there would be a wire labeled with both `x` and `r`.