Name Solution

# Digital Design using HDLs
# LSU EE 4755
# Midterm Examination

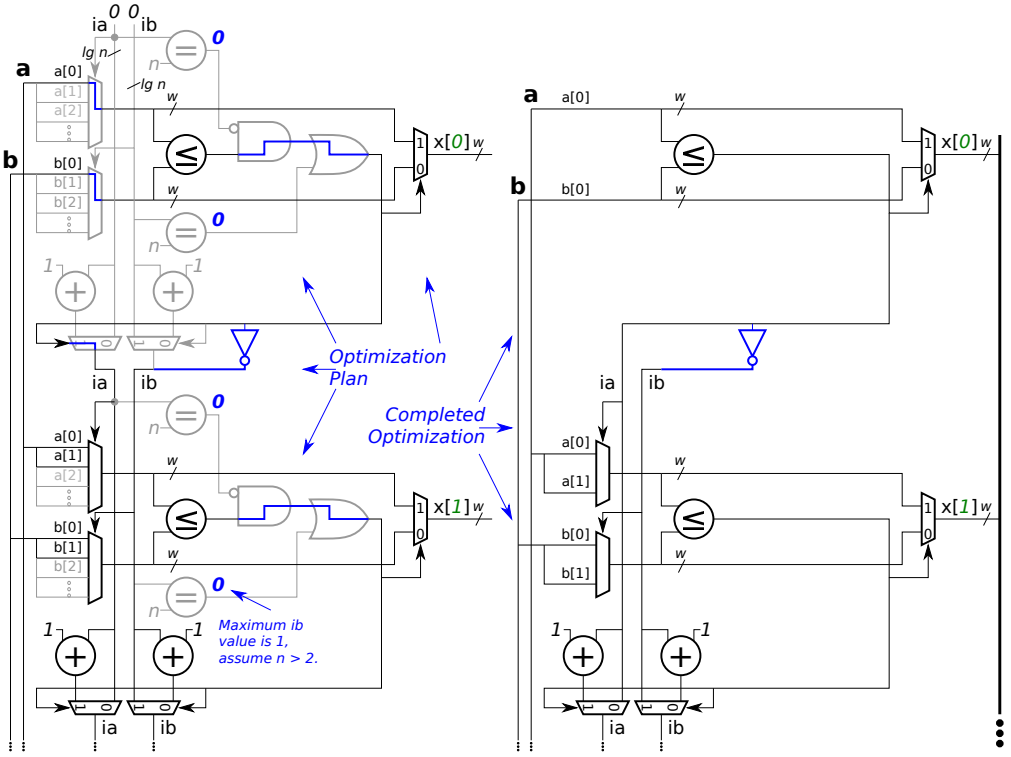
Friday, 26 October 2018 9:30–10:20 CDT

| | | |
|---|---|---|
| Problem 1 | | (22 pts) |
| Problem 2 | | (20 pts) |
| Problem 3 | | (23 pts) |
| Problem 4 | | (10 pts) |
| Problem 5 | | (25 pts) |
| Exam Total | | (100 pts) |

Alias Blockchain Apocalypse

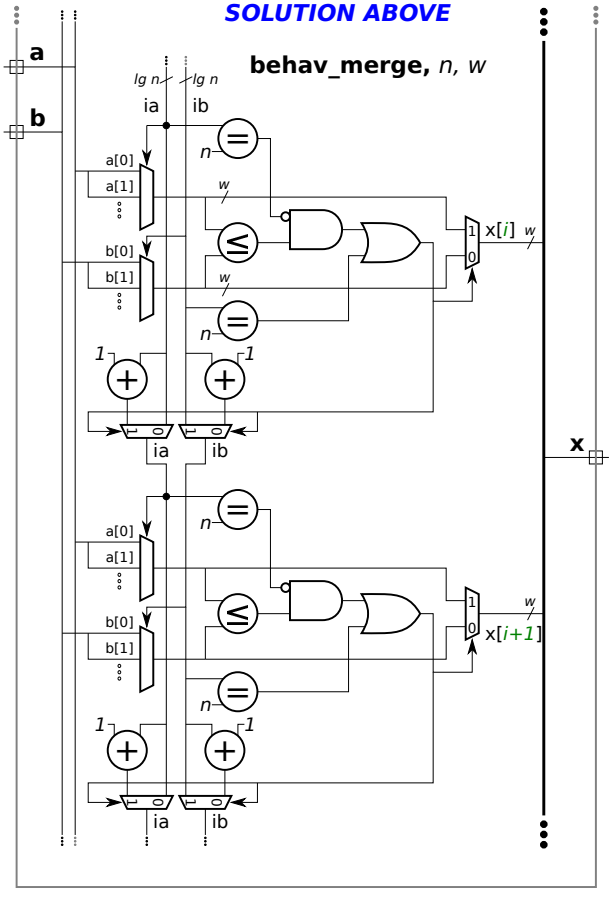
*Good Luck!*

Problem 1: [22 pts] The illustration below shows some of the inferred hardware for the `behav_merge` module from the solution to Homework 6. The hardware that's shown is for typical iterations `i` and `i+1`. Show the hardware for iterations `i=0` and `i=1` with optimizations applied.




- ☑ Show hardware for iterations `i=0` and `i=1`.
- ☑ Also show hardware for code before `for` loop.
- ☑ Optimize hardware. Take into account possible values of `ia` and `ib`.

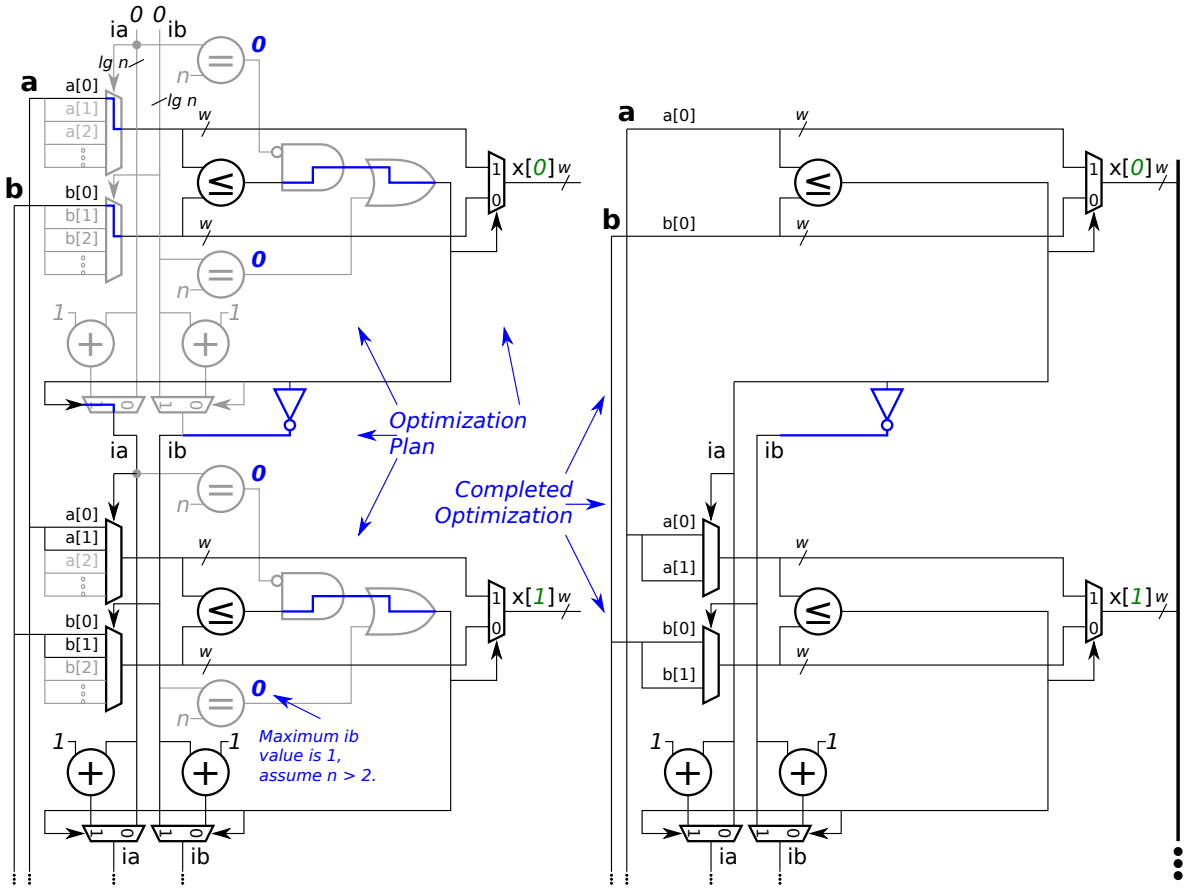
See the next page for a discussion of the solution.

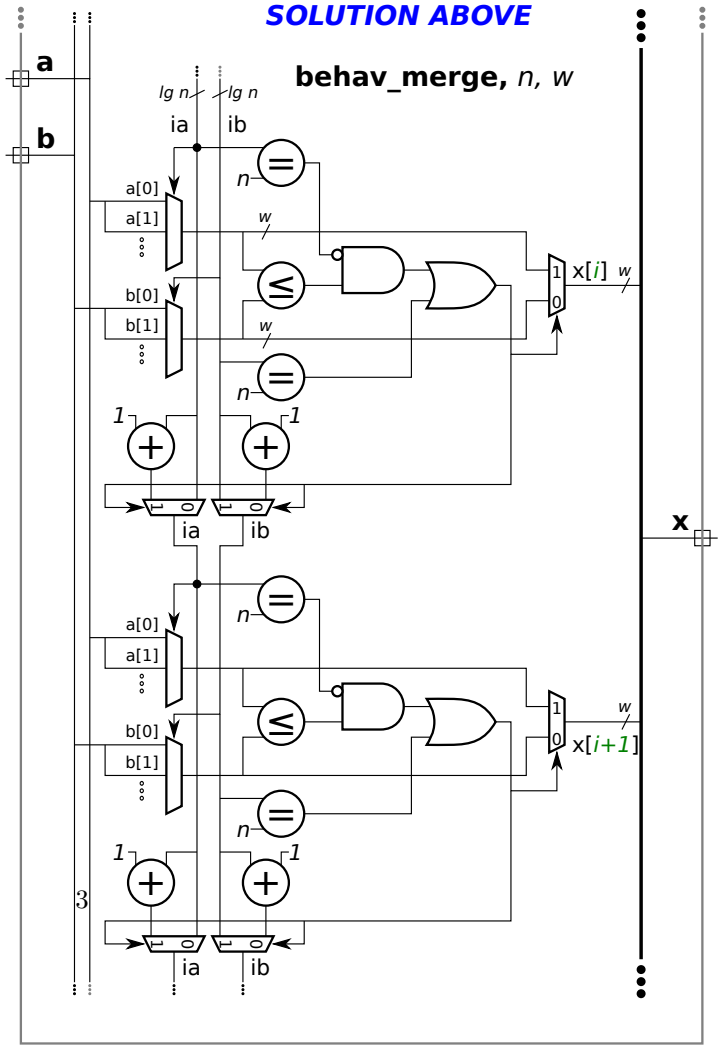



```
module behav_merge
 #( int n = 4, int w = 8 )
  ( output logic [w-1:0] x[2*n],
    input uwire [w-1:0] a[n], b[n] );

  logic [$clog2(n+1)-1:0] ia, ib;
  always_comb begin
   ia = 0; ib = 0;
   for ( int i = 0;  i < 2*n;  i++ )
     if ( ib==n || ia!=n && a[ia]<=b[ib] )
       x[i] = a[ia++]; else x[i] = b[ib++];
  end

endmodule
```




Solution appears above.

Explanation: To the left hardware that's no longer needed appears in gray. On the right the diagram is redrawn with the unneeded hardware removed. The initial zero values for `ia` and `ib` make the `a[ia]` and `b[ib]` muxen unnecessary. For `i=1` those muxen each have two inputs since the possible values for `ia` and `ib` are either 0 or 1.

A value for `n` was not given, but it is reasonable to assume that it is greater than 1. In that case the output of all of the `=n` logic blocks will be false. This makes the AND and OR gates unnecessary, and so the output of the $\leq$ block can connect directly to the `x` mux and to the logic generating the new `ia` and `ib` signals. For `i=0` the `ia` signal is equal to the output fo the $\leq$ block (that is, a 0 or 1), for `ib` (or to be exact, the least significant bit of `ib`) the output is inverted.

Problem 2: [20 pts] Appearing once again is part of the Homework 6 solution, this time with items labeled in blue. Show the cost and delay of these, as requested below. See the previous problem for the Verilog description. The phrase *most expensive* means for the value of $i$ for which the device needs all inputs, even after optimization. For the mux, show the cost and delay for the tree implementation.

☑ Cost of most expensive `a-mux` in terms of $n$ and $w$.

The mux has $n$ inputs (the size of the `a` array) of $w$ bits each. The cost is $\boxed{3w(n-1)\,\mathrm{u_c}}$.

☑ Delay of most expensive `a-mux` in terms of $n$ and $w$.

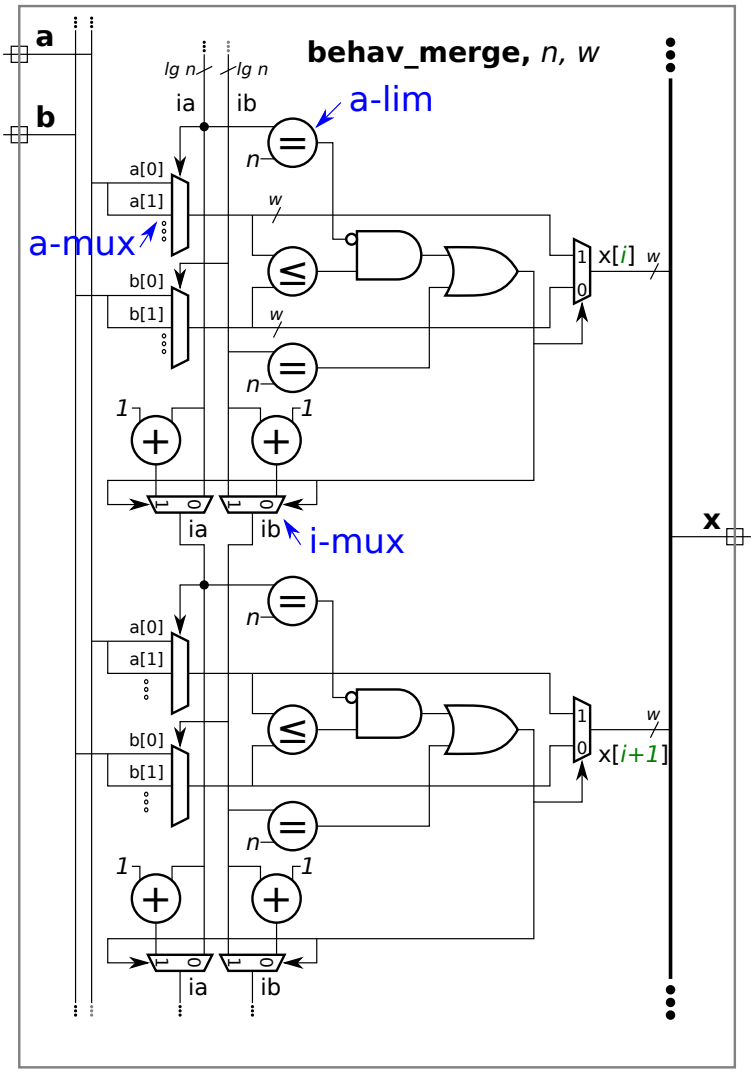
The delay is $\boxed{2\lceil \lg n \rceil\,\mathrm{u_t}}$.

☑ Cost of most expensive `i-mux` in terms of $n$ and $w$.

The `i-mux` has just two inputs of $\lg n$ bits each according to the diagram. According to the Verilog the number of bits is $\lceil \lg(n+1) \rceil$. The cost is $\boxed{3\lceil \lg(n+1) \rceil\,\mathrm{u_c}}$. Note: $3 \lg n$ would get full credit.

☑ Delay of most expensive `i-mux` in terms of $n$ and $w$.

Since there are only two inputs the delay is $\boxed{2\,\mathrm{u_t}}$.




☑ Cost of most expensive `a-lim` in terms of $n$ and $w$ ☑ after optimizing for constant inputs.

Input `n` to the equality unit is a constant, so the first column of XOR gates is replaced by NOT gates (in positions where the `n` bit is 0). So the equality module is just the NOT gates plus $n$-input AND gate, the cost of which is $\boxed{(\lceil \lg(n+1) \rceil - 1)\,\mathrm{u_c}}$.

☑ Delay of most expensive `a-lim` in terms of $n$ and $w$ ☑ after optimizing for constant inputs.

The delay is $\boxed{\lceil \lg \lceil \lg(n+1) \rceil \rceil\,\mathrm{u_t}}$.

Problem 3: [23 pts] Output `lt` of module `comp`, below, should be 1 iff `a` is strictly less than `b`, and `eq` should be 1 iff `a==b`. Both `a` and `b` are unsigned integers. The module recursively instantiates two instances of itself, one is supposed to compare the low bits of the inputs, the other compares the high bits. Complete the module so that it works for any positive `w`.

☑ Complete the module, don't miss the ☑ `FILL IN` items.

☑ Make sure that it works for odd and even values of `w`.

```
module comp
  #( int w = 8 )
   ( output uwire lt, eq, input uwire [w-1:0] a, b );

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

      assign lt = !a && b;
      assign eq = a == b;

   end else begin

      uwire llo, lhi, elo, ehi;
      localparam int wlo = w / 2;
      localparam int whi = w - wlo;

      // Instantiate two comp modules,  connect each to about half the inputs.
      //
      //     ----                  --------------      -------------- <--  ☑  FILL IN
      comp #( wlo ) clo( llo, elo, a[ wlo - 1 : 0   ],  b[ wlo - 1 : 0   ] );

      comp #( whi ) chi( lhi, ehi, a[ w - 1 : wlo   ],  b[ w - 1 : wlo ] );


      assign lt =  lhi  ||  ehi && llo ;                        <----  ☑  FILL IN

      assign eq =  elo && ehi ;                                 <----  ☑  FILL IN

   end
endmodule
```

Solution appears above, in blue, of course.

Explanation: The termination condition must be set to `w==1` because the expression `!a && b` would not set `lt` to the correct value if `a` and `b` were more than one-bit quantities. Setting `w==0` would make no sense from a functionality viewpoint.

The non-terminating case splits the bits making up the two inputs, `a` and `b`, between the two recursive instantiations, `clo` and `chi`, in a straightforward manner. Notice that `wlo` and `whi` are computed separately (rather than using `w/2` for both) to handle odd values of `w`.

Finally, outputs `lt` and `eq` must be computed from the outputs of `clo` and `chi`. Equal is the easier one. Input `a` equals `b` if their low bits and high bits are both equal. That is, `eq = elo && ehi`. For `lt` to be true either `lhi` is true (meaning that `a<b` looking only at the most-significant bits) or if the high bits are equal, `ehi`, and `llo` is true.

Problem 4: [10 pts] The output of `plus_amt`, `x`, is to be set to `b + amt`. Input `b` and output `x` are expected to be in IEEE 754 double FP format (the same format as type real). (Note: the port declarations are not to be modified in the problems below.) Several variations on the module appear below. *Hint: Solution to this problem require the correct use of* `realtobits` *and/or* `bitstoreal`. *Grading Note: The bonus problem was not on the original exam.*

(*a*) The module below does not compute the correct result. Fix the module by modifying the `always_comb` block. The module does not need to be synthesizable.

☑ Fix so that `x` is assigned the correct result, `amt` plus value of `b`.

Two solutions appears below. In the original code one operand was an integer type, `b`, the other was a real type, `amt`. In such cases the simulator would add code to convert `b` from an integer to a real. The simulator has no way of knowing that `b` already holds a value in the real format. Once `b` is converted the value is ruined. Two solutions are shown below. In the first solution two new real variables are declared, one for `b` and one for `x`. The re-interpretation system task `$bitstoreal` is used to move the value in `b` to `b_real` without changing the bits. In the statement `x_real = b_real + amt;` all three variables are real, so the simulator does not do any type conversion. Finally, `x` is assigned from `x_real` using the re-interpretation system task `$realtobits`. The second solution uses these system tasks the same way but without the intermediate variables.

```
module plus_amt
  #( real amt = 1.5 )
   ( output logic [63:0] x, input uwire [63:0] b );
   // Both x and b are IEEE 754 doubles (reals).

   real b_real, x_real;  // Declare vars to hold real values.

   always_comb begin
      b_real = $bitstoreal(b); // Re-interpret b as a real.
      x_real = b_real + amt;   // Note: Both operands are FP, so do FP add.
      x = $realtobits(x_real); // Re-interpret x_real as logic vector (int).
   end

endmodule

module plus_amt // Compact solution, avoids need for new variables.
  #( real amt = 1.5 ) ( output logic [63:0] x, input uwire [63:0] b );

   always_comb x = $realtobits( $bitstoreal(b) + amt );

endmodule
```

(*b*) [0 pts] Bonus Problem Complete the module below so that it uses the `CW_fp_add` module to do the addition. The parameters to `CW_fp_add` are already correct, just connect the inputs and outputs.

☐ Complete so that it computes the correct result.

Problem 5: [25 pts] Show the hardware that will be inferred for the Verilog code below.

☑ Clearly show module ports.

☑ Show inferred hardware. Don't optimize.

☑ Pay close attention to what is and is not inferred as a register.

```
module regs #( int w = 10, int k1 = 20, int k2 = 30 )
   ( output logic [w-1:0] y,
     input logic [w-1:0] b, c,
     input uwire clk );

   logic [w-1:0] a, x, z;

   always_ff @( posedge clk ) begin

      a = b + c;
      if ( a > k1 ) x = b + 10;
      if ( a > k2 ) z = b + x; else z = c - x;
      y = x + z;

   end

endmodule
```

Solution appears below.

Explanation: The area corresponding to the `always_ff` block is outlined in a green dashed line. Registers are shown on the right-hand boundary because the value that gets clocked into a register is the value present when control reaches the end of the block (the `end` statement above). Four values are assigned within the block, `a`, `x`, `z`, and `y`. Registers are inferred only for those variables that are a *live out* object of the block. That is true for `y` since it's also a module output and so its value is needed outside the block. In contrast, the value of `a` that is computed in the block is not used again after the `end` is reached. (When the block is re-entered a new value of `a` will be computed.) The same is true for `z`. But the value of `x` may be used after `end` is reached. That happens when the block is re-entered and $a \leq k_1$, in which case `x` is set to the previous value of `x` (the one in the register) rather than `b+10`.