LSU EE 4755 **Homework 6** Solution **Due: 10 October 2018**

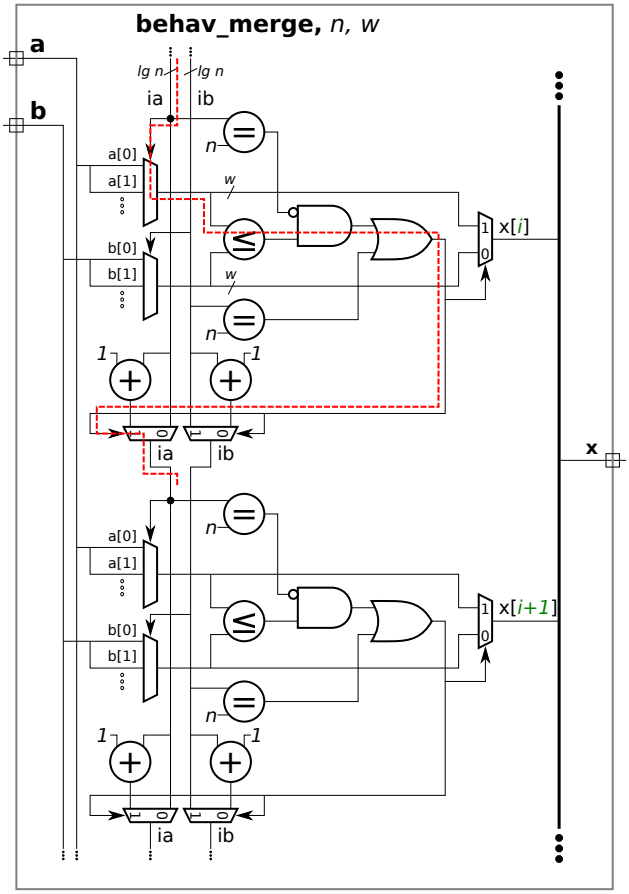
Problem 1: Use the simple model to compute the cost and delay (critical path length) of the inferred hardware for module `behav_merge` from Homework 5. This module has two inputs, `a` and `b`, each of which is an n-element sorted sequence of w-bit unsigned integer values. Output `x` is a $2n$-element array of w-bit quantities. The module assigns elements of `a` and `b` to `x` so that `x` itself is a sorted sequence of the elements from `a` and `b`.

Show the cost and delay of `behav_merge` in terms of n and w. The Homework 5 module appears below. Use the tree implementation of multiplexors for cost and delay. (See the simple model notes.) Make reasonable optimizations, such as using the same multiplexor for `a[ia]` and `a[ia++]`. Avoid tedious optimizations such as varying the number of bits in `ia` and `ib`.

Solution on next page.

```
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++ )
        x[i] = ib == n || ia < n && a[ia] <= b[ib] ? a[ia++] : b[ib++];
   end
endmodule
```



The inferred hardware appears above. The problem did not explicitly ask for the inferred hardware, but cost and delay could not be found without it. The diagram shows the hardware resulting from two `for` loop iterations, for outputs `i` and `i+1`. The cost is dominated by the cost of the multiplexors implementing `a[ia]` and `b[ib]`. Each of these muxen, before optimization, has n inputs of w bits, for a cost of $3w(n-1)\,\mathrm{u_c}$ each. Since there are $2n$ iterations, the total cost of the `a` and `b` multiplexors will be $2n \times 2 \times 3w(n-1)\,\mathrm{u_c} \approx 12wn^2\,\mathrm{u_c}$. That's expensive. The cost will be less than that because the muxen for iteration $i < n$ only need i inputs. But accounting for that would not even cut the cost in half.

The muxen producing the value of `x[i]` cost $3w\,\mathrm{u_c}$ each for a total cost of $6wn\,\mathrm{u_c}$. The muxen passing `ia` and `ib` (incremented or not) cost $3\lg n\,\mathrm{u_c}$ each for a total cost of $12n\lg n\,\mathrm{u_c}$.

Magnitude comparison units ($\boxed{\leq}$) of w bits have a cost of $4w\,\mathrm{u_c}$ and a delay of $2w+1\,\mathrm{u_t}$, so the total cost of these units is $8wn\,\mathrm{u_c}$. The $\boxed{=n}$ *limit* units test whether `ia` and `ib` have reached their maximum value, `n`. In general an ω-bit comparison unit cost $4\omega - 1\,\mathrm{u_c}$ but in this case one input is a constant, and so the first column of

XOR gates is converted into either NOT gates or wire, and so the cost is reduced to $\omega - 1\,\mathrm{u_c}$. For `behav_merge` $\omega \to \lceil \lg(n+1) \rceil \approx \lg n$. There are two limit units per iteration, for a total of $4n$ units and so their total cost is $4n \lg n\,\mathrm{u_c}$.

The adders to increment `ia` and `ib` operate on $\lg n$-bit quantities. Unoptimized and based on a ripple implementation they would cost $9 \lg n\,\mathrm{u_c}$. But since one input is the constant 1 the ripple adder can be built using binary half-adders, at a cost of $3\,\mathrm{u_c}$ per bit, for a cost of $3 \lg n\,\mathrm{u_c}$. There are $4n$ adders for a total cost of $12n \lg n\,\mathrm{u_c}$.

The cost of everything is:

$$2n\Big[\overbrace{6w(n-1)}^{2\times \text{ big mux}} + \overbrace{3w}^{\texttt{x} \text{ mux}} + \overbrace{6\lg n}^{\texttt{iab} \text{ muxen}} + \overbrace{4w}^{\boxed{\leq}} + \overbrace{2\lg n}^{2\times\boxed{=n}} + \overbrace{6\lg n}^{2\times\boxed{+1}}\Big]\,\mathrm{u_c}$$

$$= 2n\big[6wn + w + 14\lg n\big]\,\mathrm{u_c}$$

The critical path is shown as a red dashed line. (The critical path also passes through `ib`, that's omitted for clarity and because those two paths are the same length.) Assuming a tree implementation for the mux and a ripple implementation for the comparison, each section has a critical path length of $((2\lg n) + 2w + 1 + 1 + 2)\,\mathrm{u_t}$. The total critical path length is $2n[(2\lg n) + 2w + 1 + 1 + 2]\,\mathrm{u_t} \approx (4n \lg n + 4nw)\,\mathrm{u_t}$. That's long. Even if the comparison used a tree-like design with a $\lg w$ delay the critical path through the merge unit would still be very long, at least compared to the Batcher odd/even merger.

Problem 2: As was probably mentioned, a proper n-element Batcher odd/even merge module is constructed from $\frac{n}{2}\lceil \lg n \rceil$ `sort2` modules, and the critical path length through a merge module is $\lceil \lg n \rceil$ `sort2` delays.

If the previous problem was solved correctly then the cost and critical path length of `behav_merge` should be much larger than a Batcher merge. But the behavioral code in `behav_merge` has a run time of $O(2n)$ running as an ordinary program, and consumes $O(2n)$ memory, both of which are optimal for an algorithm that must operate on all of $2n$ items. In fact, recursively applied code based on `behav_merge` can sort a sequence in $O(n \lg n)$ time, which is the best one can normally get in many cases.

What is it about the hardware realization of `behav_merge` that makes it so much less efficient than the software realization? Your answer should consider how much hardware is being used at each moment in time.

In the hardware version a piece of hardware is needed for each of the $2n$ outputs. That can't be avoided because this is combinational logic. So, for example, there are $2n$ $\boxed{\leq}$ comparison units, whereas in the execution of the software version there might be one ALU with just one comparison unit which gets used $2n$ times. This kind of efficiency could be realized with sequential logic.