LSU EE 4755 **Homework 4** Solution **Due: 28 October 2020**

☣ *Paper copies will not be accepted. E-mail your solution to* `koppel@ece.lsu.edu`. *A single PDF file is preferred.*

This assignment refers to the solution to Homework 3. Pieces are shown below, the complete solution can be found at `https://www.ece.lsu.edu/koppel/v/2020/hw03-sol.v.html` *and in the directory where the original assignment was copied from.*

This solution was prepared 3 Nov 2020 at 16:23. A more detailed solution may be posted later.

Problem 1: Using the simple model compute the cost and delay of the `nnAdd` module from Homework 3 (shown below) for both `sat=0` and `sat=1`. Do so after applying optimizations for constants. Show the cost and delay in terms of w. *Hint: See the simple model notes,* `https://www.ece.lsu.edu/koppel/v/2020/lsli-simple-model.pdf`*, for the cost of a ripple adder.*

- Show cost and delay in terms of w.
- Don't forget to optimize for constant values.
- Assume that the adder will be implemented using a ripple circuit.
- Indicate both the delay of the least-significant bit of the sum and the delay of the most significant bit of the sum. *Answering this part correctly and applying it to the other problems in this assignment will reveal something important about the impact of detecting overflow and of the different methods of doing so.*

```
module nnAdd #( int w = 5, sat = 0 )
              ( output uwire [w-1:0] so, input uwire [w-1:0] a, b );
   uwire [w:0] s = a + b;
   localparam logic [w-1:0] smax = ~w'(0);
   assign so = sat && s[w] ? smax : s[w-1:0];
endmodule
```

Under the simple model the cost of a w-bit ripple adder is $9w\,\mathrm{u_c}$, the delay of the least-significant bit is $4\,\mathrm{u_t}$ and the delay of the entire sum is $2(w+1)\,\mathrm{u_t}$.

For `sat=0` the cost and delay are those of the w-bit adder described above: The cost of the `sat=0` module is $9w\,\mathrm{u_c}$, the delay of the LSB in the `sat=0` module is $4\,\mathrm{u_t}$, and the delay of the MSB in the `sat=0` module is $2(w+1)\,\mathrm{u_t}$.

When `sat=1` the overflow logic must be taken into account too. That overflow logic synthesizes into a multiplexor with the select signal connected to `s[w]`, the zero input connected to `smax`, and the one input connected to the sum, `s[w-1:0]`. Because `smax` is a constant, the cost of the multiplexor is w and the added delay is 1. So, the cost with `sat=1` is $10w\,\mathrm{u_c}$.

Because the multiplexor control signal is connected to the carry out of the adder (which would be bit position w of the sum), *all* bits of the sum must wait for the MSB to arrive. That means that the delay for all bits in the `sat=1` module is $[2(w+1)+1]\,\mathrm{u_t}$. Sure, if all you cared about was the MSB this would be no big deal. But it precludes getting a faster result with cascaded ripple adders.

There are more problems on the next pages.

Problem 2: Using the simple model compute the cost and delay of the `nnMult` module from Homework 3 for `sat=1`. Let w denote the setting of both `wa` and `wb` (they are to be set to the same value), and let y denote the setting of `wp`. Solve this for $y < 2w$. Do so after applying optimizations for constants.

Solve this using the following cost for an unsigned integer multiplier with two w-bit inputs and a $2w$-bit output: the cost using the simple model is $10w^2\,\mathrm{u_c}$ and the delay is $[8w+2]\,\mathrm{u_t}$ for the complete product and $[4i+2]\,\mathrm{u_t}$ for bit position i. (The LSB is at position $i = 0$.) (For more details on how those were derived see the comments after the Linear Multiplier in `https://www.ece.lsu.edu/koppel/v/2020/mult-seq.v.html`.)

- Show the cost and delay in terms of w and y.
- Solve this for $y < 2w$.
- Don't forget to optimize for constant values.

```
module nnMult #( int wa = 5, wb = 6, wp = wa + wb, sat = 0 )
   ( output uwire [wp-1:0] p,  input uwire [wa-1:0] a, input uwire [wb-1:0] b );

   localparam logic [wp-1:0] pmax = ~wp'(0);
   localparam int wmx = wp > wa+wb ? wp : wa+wb;
   uwire [wmx-wp:0] phi;
   uwire [wp-1:0] plo;
   assign {phi,plo} = a * b;
   assign p = sat && wp < wa + wb && phi ? pmax : plo;

endmodule
```

In order to detect overflow the multiplier must compute a $2w$-bit product. The problem statement helpfully gives the cost of such hardware as $10w^2\,\mathrm{u_c}$ and the delay as $[8w+2]\,\mathrm{u_t}$.

The only difference with the saturation logic is that it must examine $2w-y$ bits of the product. If any of those $2w-y$ bits are 1 then there is overflow. As in the previous problem there is a multiplexor with the result (product in this case) at the zero input and `pmax` at the one input. The select signal is generated by ORing the $2w-y$ high bits of the product together. The cost of the multiplexor is $y\,\mathrm{u_c}$, and the cost of the OR gate is $[2w-y-1]\,\mathrm{u_c}$. Ordinarily under the simple model the delay for an a-input OR gate would be $\lceil \lg a \rceil$. But in this case we know the less significant bits of the product arrive earlier than the more significant bits. To implement an a-input OR gate for such a situation the OR gates can be connected linearly (rather than using a reduction tree). The MSB of the product would connect to the last OR gate, and so the delay for checking whether any of the $2w-y$ bits is 1 would just be $1\,\mathrm{u_t}$.

The total $\boxed{\text{cost of } \texttt{nnMult} \text{ is } [10w^2+y]\,\mathrm{u_c}}$ and $\boxed{\text{the delay of all bits is } [8w+2+1]\,\mathrm{u_t}}$.

There are more problems on the next pages.

Problem 3: Using the simple model determine the cost and performance of module `nn1xI` (shown on the next page) for the configurations described below. In all cases, let n denote the value of `ni`, w denote the value of `ww` and `wi` (which are the same) and y denote the value of `wo`. Assume the same hardware costs as the first two problems (modifying sizes and accounting for cascading where appropriate).

(a) Find the cost (not delay in this part) for `sat=0`, `tr=0`, and $y > 2w$ (that's one configuration) and for `sat=0`, `tr=1`, and $y > 2w$ (that's a second configuration). The two costs will be very similar.

- Show the costs in terms of n, w, and y.

Short answer: The cost for `tr=0` and `tr=1` is $[10nw^2 + 9(n-1)y]\,\mathrm{u_c}$ (or lower, see detailed answer).

Detailed Answer: The $y > 2w$ condition means that the multiplier will not overflow and that all bits of the product are needed. (If y were smaller, say $y = w$, then some of the adders used to implement the multiplier would be less than w bits and so would cost less.)

The `nnMADD` module consists of both an `nnAdd` and `nnMult` module. So the cost of the `tr=0` solution will be discussed in terms of the `nnAdd` and `nnMult` modules.

For both the `tr=0` and `tr=1` cases there will be n multipliers each having two w-bit inputs. The total cost of these multipliers is $10nw^2\,\mathrm{u_c}$.

For the `tr=0` case there are n `nnAdd` units but the input to the first adder is zero (because `s[-1]=0`), so after optimization there are $n-1$ adders. When n is a power of 2, the number of adders for the `tr=1` case is $\sum_{l=0}^{(\lg n)-1} 2^l = n-1$. So the number of adders is the same in both cases.

The code instantiates y-bit adders, but good synthesis programs—and good students—will have noticed that not all adders need y bits to avoid overflow. For the `tr=0` case the `i=1` adder has two $2w$-bit inputs and so only needs to compute a sum of $2w+1$ bits to avoid overflow. So if $y > 2w+1$ the cost can be reduced by using a $2w+1$ bit adder rather than a y bit adder. Call this a *trim optimization*.

First, compute the cost without the trim optimization. The cost of each of these adders is $9y\,\mathrm{u_c}$. The total cost of the adders for both `tr=0` and `tr=1` is $9y(n-1)\,\mathrm{u_c}$.

The total cost of the `nn1xI` module without the trim optimization is $[10nw^2 + 9(n-1)y]\,\mathrm{u_c}$.

The cost with the trim optimization will be computed for `tr=1`. Let l indicate a level in the recursion tree with l corresponding to a level in which $n = 2^l$. The base case is $l = 0$, for which there are no adders. For $l > 0$ there are $n/2^l$ adders each need $2w + l$ bits, so the cost at level l is $\left[\frac{n}{2^l}9(2w+l)\right]\,\mathrm{u_c}$. The total adder cost is $\sum_{l=1}^{(\lg n)} \left[\frac{n}{2^l}9(2w+l)\right]\,\mathrm{u_c} = [9(2w+2)(n-1) - 9\lg n]\,\mathrm{u_c}$.

(b) Find the delay (not cost in this part) for `sat=0`, `tr=0`, and $y > 2w$ (that's one configuration) and for `sat=0`, `tr=1`, and $y > 2w$ (that's a second configuration). The two delays will be very different.

- Show the delays in terms of n, w, and y.
- When computing the total delay don't forget to take into account the time that inputs arrive at each port, especially for the multiplier.
- When computing total delay account for cascading of ripple units.

At launch time ($t = 0$) inputs are available at all of the multipliers. As stated in the problem, bit i is correct at time $[4i+2]\,\mathrm{u_t}$.

First consider `tr=0`. For `i=1` (the `i` from the generate loop) the two inputs to the adder are from multipliers (because for `i=0` there is no need for an adder), and so bit i arrives at $4i+2$. Because the inputs to the adder aren't all available at the same time we can't rely on the ripple adder formula for when bit i of the sum is available. We know that each BFA requires 4 units of time to compute both the sum and carry output from its inputs when those inputs are available at the same time. Therefore, bit i of that first adder is available at $4i + 2 + 4 = 4i + 6$. Accounting for

$n-1$ adders, bit i is available at $4i+2+4(n-1) = 4(i+n)-2$ and the most-significant bit, $y-1$ is available at $[4(y-1)+4n-2]\,\mathrm{u_t} = [4(y+n)-6]\,\mathrm{u_t}$.

So, the delay for the LSB when `tr=0` is $[4n-2]\,\mathrm{u_t}$ and the delay for the MSB when `tr=0` is $[4(y+n)-6]\,\mathrm{u_t}$.

For `tr=1` the computation is similar, except that the critical path passes through $\lg n$ adders rather than $n-1$ adders. Therefore the delay for bit i is $[4i+2+4\lg n]\,\mathrm{u_t}$ and the MSB is available at $[4(y-1)+2+4\lg n]\,\mathrm{u_t}$.

So, the delay for the LSB when `tr=1` is $[2+4\lg n]\,\mathrm{u_t}$ and the delay for the MSB when `tr=1` is $[4(y-1)+2+4\lg n]\,\mathrm{u_t}$.

(*c*) Find the delay for `sat=1`, `tr=0`, and $y > 2w$ (that's one configuration) and for `sat=1`, `tr=1`, and $y > 2w$ (that's a second configuration). The two delays should be very different from each other and from the delays from the previous problem.

Since $y > 2w$ there will be no saturation penalty for the multiplier. Therefore bit i of the product is stable at $4i+2$.

Because of the multiplexor, the delay though one saturating adder (an `nnAdd` module with `sat=1`) is the same for all bits. That delay is $[2(y+1)+1]\,\mathrm{u_t}$.

First consider `tr=0`. For `i=1` (the genvar, not a bit position) bit i of the sum (before saturation, and accounting for the multiplier delay) is ready at time $4i+2+4 = 4i+6$. The MSB, $y-1$, is available at $4y+2$, and the output of the mux is available at $[4y+3]\,\mathrm{u_t}$. The delay computation is different for the remaining $n-2$ adders. Consider the `i=2` (the genvar) adder. One input is from the `i=1` adder and the other is from a multiplier. The input from the `i=1` adder arrives at $[4y+3]\,\mathrm{u_t}$ (which we just calculated). By then all bits from the multiplier will have arrived. So the time at which the LSB can be computed is $4y+3$, there is no early start. The sum will be computed $2(y+1)$ later, or at a total delay of $[4y+3+2(y+1)+1]\,\mathrm{u_t} = [6y+5+1]\,\mathrm{u_t}$ including the mux. The complete sum is available at $[4y+3+(n-2)(2(y+1)+1)]\,\mathrm{u_t}$ or $[n(2y+3)-3]\,\mathrm{u_t}$.

Notice that with without saturation the time is $O(n+y)$ and that with saturation the time is $O(ny)$, much worse!

The computation is similar for the `tr=1` case. The critical path starts with a multiply, add, saturate (same as for `tr=0`) with a delay of $[4y+3]\,\mathrm{u_t}$. After that the critical path passed through $(\lg n)-1$ additional adders, so the total time is $[4y+3+(\lg n-1)(2(y+1)+1)]\,\mathrm{u_t}$ or $[(2y+3)\lg n+2y]\,\mathrm{u_t}$. Here the time is order $O(y\lg n)$ which is better than $O(ny)$ but not nearly as good as $O(n+y)$.

```
module nn1xI #( int wo = 10, wi = 4, ww = 5, ni = 2, tr = 0, sat = 0 )
   ( output uwire [wo-1:0] ao,
     input uwire [wi-1:0] ai[ni],
     input uwire [ww-1:0] wht[ni] );

   if ( tr ) begin

      if ( ni == 1 ) begin

         nnMult #(wi,ww,wo,sat) mult(ao, ai[0], wht[0] );

      end else begin

         localparam int nlo = ni / 2;
         localparam int nhi = ni - nlo;
         uwire [wo-1:0] aolo, aohi;
         nn1xI #(wo,wi,ww,nlo,1,sat) nnlo(aolo, ai[0:nlo-1], wht[0:nlo-1]);
         nn1xI #(wo,wi,ww,nhi,1,sat) nnhi(aohi, ai[nlo:ni-1], wht[nlo:ni-1]);
         nnAdd #(wo,sat) add(ao,aolo,aohi);

      end

   end else begin

      uwire [wo-1:0] s[ni-1:-1];
      assign s[-1] = 0;
      assign ao = s[ni-1];

      for ( genvar i = 0;  i < ni;  i++ )
        nnMADD #(ww,wi,wo,sat) madd( s[i], wht[i], ai[i], s[i-1] );

   end

endmodule

module nnMADD #( int wa = 10, wb = 5, ws = wa + wb, sat = 0 )
   ( output uwire [ws-1:0] so,
     input uwire [wa-1:0] a, input uwire [wb-1:0] b, input uwire [ws-1:0] si);

   uwire [ws-1:0] p;
   nnMult #(wa,wb,ws,sat) mu(p, a, b);
   nnAdd #(ws,sat) ad(so, si, p);

endmodule
```

There are even more problems on the next pages.

Problem 4: Consider module `nnOxI` instantiated with `no=1`, `tr=0`, for both `sat=1` and `sat=2`. (A slightly simplified version appears below.) Let n denote the value of `ni`, w denote the value of `wi` and `ww` (which are the same), and let y denote the value of `wo`.

Assume that $2w < y < \lceil \lg n(2^w - 1)^2 \rceil$. That is, y is large enough so that the multipliers can't overflow but not so large that the adders can't overflow.

(a) Compute the cost and delay for both the `sat=1` and `sat=2` cases. For `sat=1` just re-use answers from the previous problems.

- Show answers in terms of n, w, and y.
- Don't forget that the value of `wo` in the `nn1xI` instantiations depends upon `sat`.

When `sat=1` and `no=1` the hardware for `nnOxI` is the same as that of `nn1xI`.

The cost of the `sat=0` instantiation based on the answer to Problem 3 is $[10nw^2 + 9(n-1)y]\,\mathrm{u_c}$. The cost of the saturation hardware is that of $n-1$ 2-input, y-bit multiplexors in which one input is a constant. The cost of these is $[(n-1)y]\,\mathrm{u_c}$. So the total cost for `nnOxI` with `sat=1` is $[10nw^2 + 9(n-1)y + (n-1)y]\,\mathrm{u_c} = [10nw^2 + 10(n-1)y]\,\mathrm{u_c}$.

The delay has been computed in Problem 3, it is $[n(2y+3) - 3]\,\mathrm{u_t}$.

When `sat=2` the `nn1xI` modules are instantiated with `sat=0`, and so their cost and delay are $[10nw^2 + 9(n-1)r]\,\mathrm{u_c}$ and $[4(r+n) - 6]\,\mathrm{u_t}$ where r is the value of `wr` for which they were instantiated. Note that $r = \lceil \lg n(2^w - 1)^2 \rceil$.

Module `nnOxI` checks for saturation by checking whether the high $r - y$ bits of `ar` are non-zero. That can be done using an OR gate, with a cost of $[r - y - 1]\,\mathrm{u_c}$. If the OR used a tree reduction the delay would be $\lg(r-y)\,\mathrm{u_t}$, but if we expect bit $i+1$ to arrive at least one $\mathrm{u_t}$ later than bit i a linear connection of OR gates would be faster, and have a net delay of just 1. So the total delay is $[4(r+n) - 6 + 1]\,\mathrm{u_t}$.

The total cost includes a 2-input, y-bit multiplexor and the $(r-y)$-input OR gate. One input to the mux is constant, so its cost is y. The total cost with this hardware is $[10nw^2 + 9(n-1)r + y + (r-y-1)]\,\mathrm{u_c}$ or $[10nw^2 + 9(n-1)r + r - 1]\,\mathrm{u_c}$ where $r = \lceil \lg n(2^w - 1)^2 \rceil$.

(b) In terms of the costs computed above is `sat=2` always better, always worse, or sometimes better than `sat=1`? Be specific of course.

Recall that for `sat=1` the cost is $C(1,n,w,y) = [10nw^2 + 10(n-1)y]\,\mathrm{u_c}$ and for `sat=2` the is $c(2,n,w,y) = [10nw^2 + 9(n-1)r + r - 1]\,\mathrm{u_c}$.

To solve this compute $C(1,n,w,y) - C(2,n,w,y)$. If the result is always positive then `sat=2` always costs less, etc.

$C(1,n,w,y) - C(2,n,w,y) = 10(n-1)y - 9(n-1)r - r + 1$

Recall $r = \lceil \lg n(2^w - 1)^2 \rceil$ and that the assumption is that $2w < y < r$. We can approximate $r \approx 2w + \lg n$.

The cost benefit for `sat=2` is less favorable larger when y is smaller. Consider one minus the smallest value of y, which is $y = 2w$ Then $C(1,n,w,y) - C(2,n,w,y) = 10(n-1)2w - 9(n-1)(2w + \lg n) - (2w + \lg n) + 1 = (n-1)2w - 9(n-1)\lg n - 2w - \lg n + 1 \approx (n-1)2w - 9(n-1)\lg n$. This expression is positive when $w > 2.25 \lg n$. Generally when w is large `sat=2` works better, when n is large `sat=1` works better.

```
module nnOxI #( int no = 4, ni = 2, wo = 10, wi = 4, ww = 5, tr = 0, sat = 0 )
   ( output uwire [wo-1:0] ao[no],
     input uwire [wi-1:0] ai[ni],   input uwire [ww-1:0] wht[no][ni] );

   // Compute number of bits to represent largest possible value that
   // can appear on an ao.
   localparam int wr = $clog2( ( 2**wi - 1 ) * ( 2**ww  - 1 ) * ni );

   if ( sat < 2 ) begin
```

```
    for ( genvar i = 0;  i < no;  i++ )
      nn1xI #(wo,wi,ww,ni,tr,sat) row( ao[i], ai, wht[i] );

  end else begin

    for ( genvar i = 0;  i < no;  i++ ) begin

      uwire [wr-1:0] ar;
      nn1xI #(wr,wi,ww,ni,tr,0) row( ar, ai, wht[i] );
      assign ao[i] = ar[wr-1:wo] ? ~wo'(0) : ar[wo-1:0];

    end

  end

endmodule
```

Problem 5: *Zero points will be given for the answer to this question, but please try your very best to answer it.* Suggest a method of saturating `ao` that avoids the extra `wo` bits needed (for `nn1xI`) when `sat=2` but also avoids the critical-path-killing saturation logic used when `sat=1`. Your solution could add extra ports to all modules except `nn0xI`. A correct solution would detect overflow under the same conditions as `nn0xI` does with `sat=1`.