**LSU EE 3755** **Homework 3** Solution **Due: 9 October 2013**

**Problem 1:** A Verilog description of yet another population count module appears below.

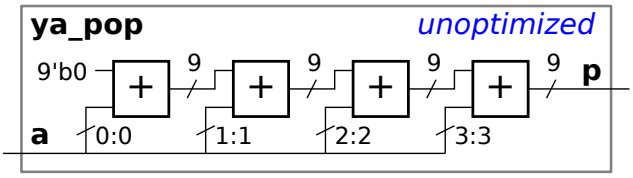
```
module ya_pop(p,a);
   parameter N = 256;
   input [N-1:0] a;
   output        p;
   reg [8:0]     p;
   integer       i;

   always @( a ) begin
      p = 0;
      for ( i=0; i<N; i = i+1 ) p = p + a[i];
   end

endmodule
```

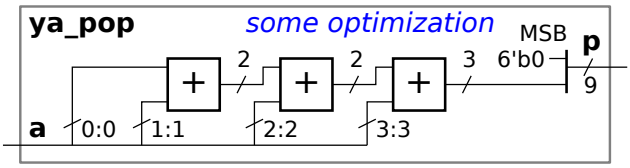
(*a*) Sketch the synthesized hardware when `N` is 4.

Several versions of the synthesized hardware will be shown. The first shows hardware that is directly inferred from the Verilog without any optimization:

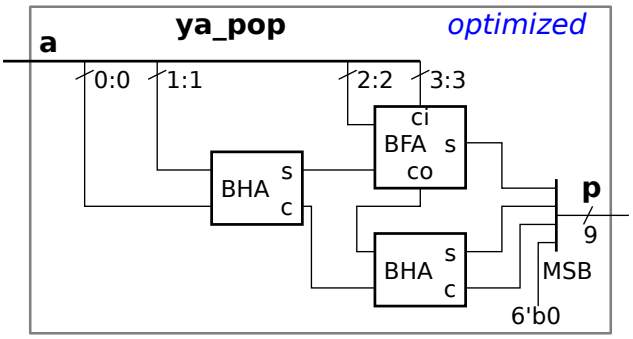



Note that in the code Verilog `p` is declared as a register, the illustration above only shows the output labeled as `p`, but the upper input to each adder is also `p`. Each assignment to `p` in the Verilog code (including the multiple loop iterations) creates a wire, so there are five different `p`s.

Note that in this hardware the first (leftmost) adder has a constant input of zero. Obviously that adder is not needed since the output will be the same as the input. Also note that all adder outputs are nine bits, because that's how `p` was declared. But the maximum possible sum, 4, only needs three bits to represent, and that's only for the last adder. These ideas are used in the version below:




Note that the lower input to each adder is just one bit, and so a complete adder is unnecessary. In the final version this simplification is applied, we end up with a circuit using just a binary full adder (BFA) and two binary half adders (BHAs):




Grading Note: Full credit would be given for any of these designs. That said, students are expected to know what a BFA and a BHA is and how the optimized circuit above works.

(*b*) Estimate the cost in terms of gates when `N` is 4 assuming that a ripple adder was inferred for the `+` operator and that reasonable optimizations were made. For this problem assume that an XOR, and other basic gates have a cost of 1.

As described the previous part, the adder logic for the four-bit population counter can be simplified to something very different than the series of ripple adders that would initially be inferred from the Verilog code. Nevertheless, to keep things simple this part asks for a solution that retains the series of ripple adders with only reasonable optimizations. In this solution *reasonable optimizations* will mean that the number of bits in each adder is reduced to that which is needed for the maximum possible output value and that the initial adder can be discarded.

Given the cost model, the cost of a binary full adder (BFA) is 5 and so the cost of an $q$-bit ripple adder is $5q$. For the $N = 4$ case we need two 2-bit ripple adders and one 3-bit ripple adder, for a total cost of $5 \times 2 \times 2 + 5 \times 3 = 35$.

For the $N$-bit case first assume that all adders are 9 bits. The cost would be $5 \times 9N = 45N$. Let $n = \lceil \lg(N+1) \rceil \approx \lg N$, the number of bits needed in the output of the last adder. Using this a closer approximation of the cost can be obtained, $5(\lg N)N$. For a tighter approximation note that a $j$-bit adder can be used in each of $2^{j-1}$ stages. For example, a 3-bit adder can be used in $2^{3-1} = 4$ stages, starting at the stage where the sum can reach 4 up until the stage where the sum can reach 7. Summing over each adder size we can an obtain the cost of the entire population counter. If $N$ is a power of 2 the cost will be $\boxed{5 + \sum_{j=1}^{\lg N} 5i2^{i-1}}$, where the 5+ is needed for the extra bit in the last stage.

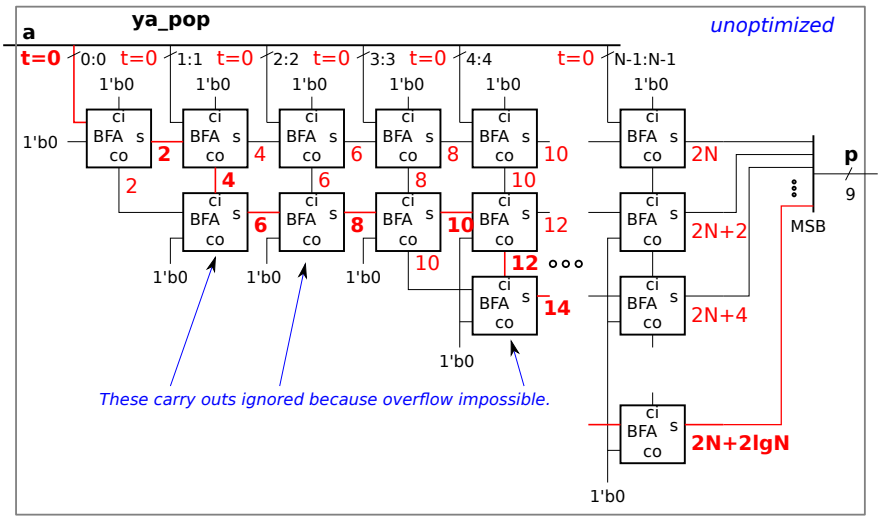
A natural question that might be troubling someone reading this is "Did I need to write all that to get full credit?!?" The answer is, no. For full credit one did have to recognize that most of the $N-1$ stages contain an adder of **more than one bit** and so consisted of more than one binary full adder per stage and so each stage had a cost of more than 5 gates.

(*c*) Making the same assumptions as in the previous part, determine the delay for arbitrary `N` and for `N=4`. In particular assume that optimizations were made to reduce cost but that the circuit was not re-organized to improve performance (see the next part). In your delay model use 2 for the delay of an XOR and 1 for other gates.

The delay from any input to any output of a BFA or BHA is 2 units. The critical path for the $N = 4$ optimized population counter illustrated above goes through all 3 adders, and so the delay is 6.

The solution for arbitrary $N$ is for a circuit using a ripple adder for each iteration (or stage). The **wrong** approach is to add up the delays of each of the ripple adders. Under the delay model a $j$-bit ripple adder has a delay of $2j$. For $N$ stages of 9-bit ripple adders the delay would be $2 \times 9N$ units of delay. This is wrong both because the adders in earlier stages can be smaller and because an adder in stage $i$ can start before the adder in stage $i-1$ completely finishes.

The diagram below shows the correct approach to computing delay, with the critical path shown in **bold red**. First note the size of the ripple adder in each stage. The first stage's ripple adder consists of just one BFA, the subsequent 3 stages need only 2 BFA. The stage receiving $a_3$ still has just 2 BFA's because the maximum value can be 4, and that can be represented using two sum bits and a carry out.




Next notice that the critical path flows through the bottommost BFAs (the delay is shown in red numbers, critical path delays are **bold red**). The path flows through $N$ BFAs to the right and $\lg N$ BFAs in the down direction, for a total path length of $N + \lg N$. The delay is 2 units through each BFA, for a total delay of $\boxed{2(N + \lg N)}$.

The synthesized hardware above (with the timing analysis) uses binary full adders. Because every BFA has one of its inputs connected to a logic 0, they can be replaced by binary half adders. This change would reduce cost but not affect the timing. Another easy optimization is use each ripple adder for two bits of $a$, one going in the main input, the other in the carry in.

**Problem 2:** As the previous problem hinted, it is possible to obtain faster performance from the population counter (than is possible from a synthesis program that does not apply the technique described below). Faster performance can be obtained by using use several smaller population counters and adding their sums.

($a$) Briefly describe why this would result in faster performance.

Based on the solution to the previous part, the time the circuit takes to compute the population of an $N$-bit value is $2(N + \lg N)$. Suppose two $N/2$ population counters were used. Each would take $2(N/2 + \lg N - 1)$, but since they would be working at the same time there would be no need to add their delays. Adding the sum of each using a ripple adder would take an additional $2 \lg N$ time units. The total delay using 2 population counters would be $2(N/2 + \lg N - 1) + 2 \lg N = 2(N/2 + 2 \lg N - 1)$ which is almost half $2(N + \lg N)$ for $N = 256$.

Note: For this simple circuit the Cadence Encounter RTL Compiler Version RC10.1.306 - v10.10-s357_1 synthesis program was able to synthesize fast hardware for all versions of the Verilog provided in this problem and solution. That is, the optimization that we hand-applied was unnecessary for performance. The point of the assignment was to learn how to use Verilog and to learn possible ways to transform designs to obtain higher performance, for those times when the synthesis program won't figure it out.

($b$) Write a Verilog behavioral description of a 256-bit population count module which instantiates four 64-bit `ya_pop` modules.

Solution appears below. The #64 between the module name and instance name specifies the parameter value. The module uses implicit behavior code to compute the final sum, which in this case is slightly more convenient than behavioral code.

```
// SOLUTION
module pop256x4(p,a);
   input [255:0] a;
```

```
   output        p;
   wire [8:0]    p;
   wire [8:0]    p0, p1, p2, p3;

   ya_pop #64 pop0( p0, a[63:0]    );
   ya_pop #64 pop1( p1, a[127:64]  );
   ya_pop #64 pop2( p2, a[191:128] );
   ya_pop #64 pop3( p3, a[255:192] );

   assign        p = p0 + p1 + p2 + p3;

endmodule
```

(*c*) Write a Verilog behavioral description of an `N`-bit population count module with parameters `N` and `M`, where `N` is a multiple of `M`. Unlike the module from the previous part, the module for this part should not instantiate any other modules. In procedural code it should determine the population in `M` parts and add them together. Solution to this problem may require Verilog's equivalent of arrays, called *memories*. See page 46 of the Verilog XL manual linked to the course's references page, `http://www.ece.lsu.edu/ee3755/ref.html`.

The solution appears below. (It does not use memories.) Our goal was to make sure that each of the `M` population counters start at the same time. That is, one should not wait for another to finish. This is ensured below by initializing the running sum, `pa`, to zero for each iteration of `j`.

```
module red_pop(p,a);
   parameter N = 256;
   parameter M = 4;   // Number of parts.
   parameter N_PER_PART = N/M;
   input wire [N-1:0] a;
   output reg [8:0] p;
   reg [8:0]    pa;
   integer      i, j;

   always @( a ) begin
      p = 0;
      for ( j=0; j<M; j=j+1 ) begin

         // Compute the population of a part.
         pa = 0;
         for ( i=0; i<N_PER_PART; i=i+1 ) pa = pa + a[ j*N_PER_PART + i ];

         // Add the population to a running total.
         p = p + pa;
      end
   end
endmodule
```