6.004 Tutorial Problems
L09 – Combinational Logic 2

Note: A small subset of essential problems are marked with a red star (★). We especially encourage you to try these out before recitation.

Problem 1. ★

We want to implement a parametric function `reverse#(n)` that reverses the bits of its n-bit input argument. For example, if `Bit#(4) x = {a, b, c, d}`, then `reverse#(4)(x)` should return `{d, c, b, a}`.

(A) Implement `reverse#(n)` by recursing on the parameter `n` (i.e., calling `reverse#(k)` with `k < n`). You cannot use a `for` loop.

```
function Bit#(n) reverse#(Integer n)(Bit#(n) x);




endfunction
```

(B) Implement `reverse#(n)` using a `for` loop.

```
function Bit#(n) reverse#(Integer n)(Bit#(n) x);




endfunction
```

Problem 2.

Parameterize the bit-scan-reverse function from Lab 3 to take as input an n-bit vector and output the index of the first non-zero bit scanned from the largest index (i.e., the position of the most-significant 1). **Assume that the parameter n is a power of 2.**

(A) Implement `bitScanReverse#(n)` without using a `for` loop.

```
function Bit#(log2(n)) bitScanReverse#(Integer n)(Bit#(n) x);




endfunction
```

(B) Implement `bitScanReverse#(n)` using a `for` loop.

```
function Bit#(log2(n)) bitScanReverse#(Integer n)(Bit#(n) x);




endfunction
```

(C) When synthesized manually (i.e., without logic optimizations, just the gates that your circuit expresses), how does propagation delay grow with the number of input bits for each implementation? Use order-of notation.

Problem 3. ★

In Lab 3, we wrote a function isPowerOfTwo that computes whether a 4-bit input was a power of 2 or not. This function checks whether there is only one bit in the input that is equal to 1.

We want to generalize isPowerOfTwo by rewriting it as a parametric function that works with inputs of arbitrary bit-width.

(A) Implement isPowerOfTwo#(n) using a for loop. Do not use addition to count up the bits of the input, which would be inefficient.

```
function Bool isPowerOfTwo#(Integer n)(Bit#(n) x);




endfunction
```

(B) If your implementation above has $\Theta(n)$ propagation delay when synthesized manually (i.e., without logic optimizations, just the gates that your circuit expresses), then rewrite isPowerOfTwo#(n) so that it has $\Theta(\log n)$ propagation delay.

Hint: Since you used a for loop above, you probably have a linear chain of gates in your design. Instead, think about how to solve this problem by composing functions so that, at each step, you halve the number of input bits each function processes. This will produce a tree of gates with logarithmic depth. You'll likely need to use an auxiliary parametric function that recurses on its own parameter.

Problem 4. (adapted from Quiz 1 Fall 2018, 20/100 points)
(NOTE: Part C of this question will be much easier once you have completed Lab 4)

(A) (5 points) The following parametric function f performs a basic operation using a and b. We want f2 to implement the same function as f. Fill in the blank in f2 to make the two functions equivalent. Write a single-line expression that uses the ternary operator (? :).

```
function Bit#(n) f#(Integer n)(Bit#(n) a, Bit#(1) b);
    Bit#(n) x = 0;
    for (Integer i = 0 ; i < n ; i = i+1) begin
            x[i] = a[i] ^ b;
    end
    return x;
endfunction

function Bit#(n) f2#(Integer n)(Bit#(n) a, Bit#(1) b);
    return (          ) ?        :        ;
endfunction
```

(B) (5 points) Write the truth table for the combinational device described by the function below.

```
function Bit#(2) f(Bit#(1) a, Bit#(1) b, Bit#(1) c);
    Bit#(2) ret = zeroExtend(a) + signExtend(b);
    case ({a,b})
            0: ret = {1, c};
            2: ret = {a ^ b, a & b};
            3: ret = ~signExtend(c);
    endcase
    return ret;
endfunction
```

a	b	c	ret[1]	ret[0]
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

(C) (5 points) The following parametric function g performs a specific arithmetic operation on n-bit operands a and b. We want the function g2 to implement g in a single line of code. Fill in the blank with a single expression to make g2 equivalent to g.

```
function Bit#(1) g#(Integer n)(Bit#(n) a, Bit#(n) b);
   Bit#(2) ret = 'b10;
   for (Integer i = n-1 ; i >= 0 ; i = i-1) begin
  if ({a[i], b[i]} == 'b01) ret = {0, ret[1] | ret[0]};
  else if ({a[i], b[i]} == 'b10) ret = {0, ret[0]};
end
return ret[1] | ret[0];
endfunction

function Bit#(1) g2#(Integer n)(Bit#(n) a, Bit#(n) b);

   return _____;

endfunction
```

(D) (5 points) Finish the following circuit diagram to implement function computeB, given below. You may only use 32-bit 2-to-1 multiplexers, constants (0, 1, 2, 3, ...) and logic gates (AND, NOT, OR, XOR). We have provided three 32-bit greater-than-or-equal (>=) comparators for you.

```
function Bit#(32) computeB(Bit#(32) in);
  Bit#(32) out = 0;
  if ( in >= 1 ) out = 1;
  if ( in >= 5 ) out = 5;
  if ( in >= 10 ) out = 10;
  return out;
endfunction
```