**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.

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

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

```plaintext
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 \( \text{bitScanReverse}(n) \) without using a \texttt{for} loop.

\[
\text{function Bit}(2\log_2(n)) \text{ bitScanReverse}(\text{Integer } n)(\text{Bit}(n) x);
\]

\[
\text{endfunction}
\]

(B) Implement \( \text{bitScanReverse}(n) \) using a \texttt{for} loop.

\[
\text{function Bit}(2\log_2(n)) \text{ bitScanReverse}(\text{Integer } n)(\text{Bit}(n) x);
\]

\[
\text{endfunction}
\]

(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) ★

(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
```

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>ret[1]</th>
<th>ret[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(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.

```plaintext
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.

```plaintext
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
```
Problem 5. Taken from Quiz 2, Fall 2020 Problem 2

(A) (2 points) Consider a function, mod3, that takes an unsigned 2-bit input x and returns a 2-bit result which is equal to x modulo 3. Your result should be in the range of {0, 1, 2}. Fill in the truth table below so that it describes the correct behavior of this function.

(label: 2A)

<table>
<thead>
<tr>
<th>x[1]</th>
<th>x[0]</th>
<th>out[1]</th>
<th>out[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(B) (3 points) Implement the mod3 function in Minispec by filling in the function definition below. **Your code should describe a circuit that when synthesized manually without optimizations results in at most 4 (one or two input) logic gates.** Your solution can use the following gates: inverter, 2-input AND, OR, NAND, NOR, or XOR gates. (Hint: The bitwise logical operations in Minispec are: ~ (NOT), & (AND), | (OR), ^ (XOR)).

(label: 2B)

```minispec
function Bit#(_____) mod3 (Bit#(_____) x);

    return ______________________________;

endfunction
```

(C) (2 points) Manually synthesize your function into a combinational circuit.

(label: 2C)
(D) (8 points) Complete the truth table for the following Minispec function.

```minispec
function Bit#(3) f(Bit#(3) a);
    Bit#(3) ret = 3'b100;
    case (a[2], a[0])
        0: ret = {1'b0, a[1]^a[0], 1'b1};
        1: ret = signExtend(a[1]) & ret;
        3: ret = {a[0], ~a[2:1]};
        default: ret = 3'b001;
    endcase
    return ret;
endfunction
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 6: from Quiz 2, Fall 2020

For an unsigned \( n \)-bit number \( x \),

\[
x \mod 3 = (x[0] - x[1] + x[2] - \ldots + (-1)^n x[n]) \mod 3.
\]

Complete the following Minispec function \( \text{mod3} \) that takes an unsigned \( n \)-bit input \( x \) and returns a 2-bit result equal to \( x \) modulo 3 in the range of \{0,1,2\}.

```verilog
define function mod3(#(____)) (____ x);
  if (n==1) return ______;
  else begin
    Integer h=n/2;
    Bit#(2) lower = mod3#(h)(x[h-1:0]);
    Bit#(2) upper = mod3#(n-h)(x[n-1:h]);
    if ((h&1)==1) begin
      if ((lower==0) && (upper==1)) return ______;
      else return ______;
    end else begin
      if ((lower==2) && (upper==1)) return ______;
      else return ______;
    end
  end
endfunction
```