6.004 Worksheet Questions
L09 – Combinational Logic 2

Note: A 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);


      return (n == 1)? x : {x[0], reverse#(n-1)(x[n-1:1])};


endfunction
```

The recursive algorithm here states that the reverse of an n-bit number is the LSB (x[0]) concatenated with the reverse of the n-1 most significant bits. The curly braces indicate concatenation. The ternary statement provides the base case: the reverse of a single bit is just the bit itself. Otherwise, we return the recursive definition.

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

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

      Bit#(n) res = 0;
      for (Integer i = 0; i < n; i = i + 1)
            res[i] = x[n-1-i];
      return res;

endfunction
```

We first create a new n-bit wire and describe the connections it makes iteratively in a for loop. Note that each iteration of the for loop corresponds to a wire connecting the i-th bit of res to the (n-1-i)-th bit of x. Remember that for loops can generate new hardware on each iteration.

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 and n >= 2.**

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

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

    if (n == 2) return x[1];
    else begin
        let upper = bitScanReverse#(n/2)(x[n-1:n/2]);
        let lower = bitScanReverse#(n/2)(x[n/2-1:0]);
        return (upper == 0 && x[n / 2] == 0)?
            {1'b0, lower} : {1'b1, upper};
    end


endfunction
```

NOTE: You could also avoid the if-else statement by defining the base case **`function Bit#(1) bitScanReverse#(2)(Bit#(n) x) = x[1];`** separately.

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

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


    Bit#(log2(n)) res = 0;
    for (Integer i = 0; i < n; i = i + 1)
        res = (in[i] == 1) ? i : res;
    return res;


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.

Without a loop (A), delay grows with Θ(log n), as the implementation uses a balanced tree of `bitScanReverse#()` functions (at each step, each additional function processes half of the bits, and the upper and lower halves are computed in parallel).

With a loop (B), delay grows with Θ(n), as the implementation uses a chain of muxes on res.

(Different answers are possible, depending on your implementations.)

Problem 3. ★

In Lab 3, we write a function `isPowerOfTwo` that computes whether a 4-bit input is 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);
      Bool someOnes = False;
      Bool twoOrMoreOnes = False;
      for (Integer i = 0; i < n; i = i + 1) begin
            if (x[i] == 1) begin
                  twoOrMoreOnes = someOnes;
                  someOnes = True;
            end
      end
      return someOnes && !twoOrMoreOnes;
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.

```
Typedef enum { ZeroOnes, OneOne, TwoOrMoreOnes } Pow2Res;

function Pow2Res pow2#(1)(Bit#(1) x);
      return (x == 1)? OneOne : ZeroOnes;
endfunction

  function Pow2Res pow2#(Integer n)(Bit#(n) x);
      let lower = pow2#(n/2)(x[n/2-1:0]);
      let upper = pow2#(n-n/2)(x[n-1:n/2]);
      return (lower == ZeroOnes && upper == ZeroOnes)? ZeroOnes :
             ((lower == OneOne && upper == ZeroOnes) ||
             (lower == ZeroOnes && upper == OneOne))? OneOne :
                                                      TwoOrMoreOnes;
endfunction

function Bool isPowerOfTwo#(Integer n)(Bit#(n) x);
      return pow2#(n)(x) == OneOne;
endfunction
```

Problem 4. From Past Quizzes (Fall 2018) ★

(A) 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 (     b==1       ) ?     ~a       :       a       ;
endfunction
```

Function f will return the input a with every digit XOR-ed with b. If b = 0, then the XOR does not change anything (a ^ 0 = a). If b = 1, then XOR will flip each bit (a ^ 1 = ~a). Therefore, we can write a ternary operator that is conditional on the value of b. Notice that we cannot directly put b into the conditional part of the operator. The conditional expects a Bool type and b is a 1-bit wire. We convert the wire into a Bool type by testing for equality with 1 (`b == 1`).

(B) 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	**1**	**0**
0	0	1	**1**	**1**
0	1	0	**1**	**1**
0	1	1	**1**	**1**
1	0	0	**1**	**0**
1	0	1	**1**	**0**

1	1	0	**1**	**1**
1	1	1	**0**	**0**

We go through each case statement one at a time to fill out the truth table:

- If a = 0 and b = 0, then we are in the first two rows of the truth table. The output (ret) is {1, c} which indicates a 2-bit output where the LSB is c and the MSB is 1. Thus ret[1] = 1 and ret[0] = c.
- In the case where {a,b} = 2, we know that a = 1 and b = 0. Therefore, the return value is {1 ^ 0, 1 & 0} = {1, 0}. The output does not depend on c, so the two corresponding rows of the truth table are identical.
- When {a,b} = 3, ret is the inverse of the sign-extended value of c. Here, we are operating in the last two rows of the truth table since a = 1 and b = 1. In the first row, where c = 0, signExtend(c) = 2'b00, so the inverse 2'b11 is in the truth table. In the other case, where c = 1, signExtend(c) = 2'b11 and we see the inverse 2'b00 in the truth table.
- If the input does not fall into any case statement (the third and fourth rows of the truth table), then the output is zeroExtend(a) + signExtend(b). zeroExtend(a) extends a from one bit to two bits (i.e. from 1'b0 to 2'b00). signExtend(b) will extend the MSB of b to become 2'b11. The sum of these two values is 2'b11 which gives us the third and fourth rows of the truth table.

(C) 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  (a <= b) ? 1 : 0                    ;

endfunction
```

We are told that g performs an arithmetic operation on a and b, so we will first try to figure out what it is doing. The logic is generated using a for loop going from the MSB of the inputs down to the LSB. Breaking down the logic, we can see the following:

- The MSB of ret indicates if a[n-1 : i] and b[n-1 : i] are equal. If the a[i] and b[i] are different, then ret[1] = 0 since they differ at this point. Otherwise, ret[1] remains unchanged.
- The LSB of ret indicates if a[n-1: i] < b[n-1 : i]. In the first branch of the if statement, we see that a[i] < b[i] (since a[i] = 0 and b[i] = 1). This indicates that a[n-1 : i] < b[n-1 : i]

only if the inputs have been equal thus far (as indicated by ret[1]) or the a has already been determined to be less than b (as indicated by ret[0]). In the else-if branch, we see similar logic which sets ret[0] to be 1 only if ret[0] was already one (i.e. a was already determined to be less than b)

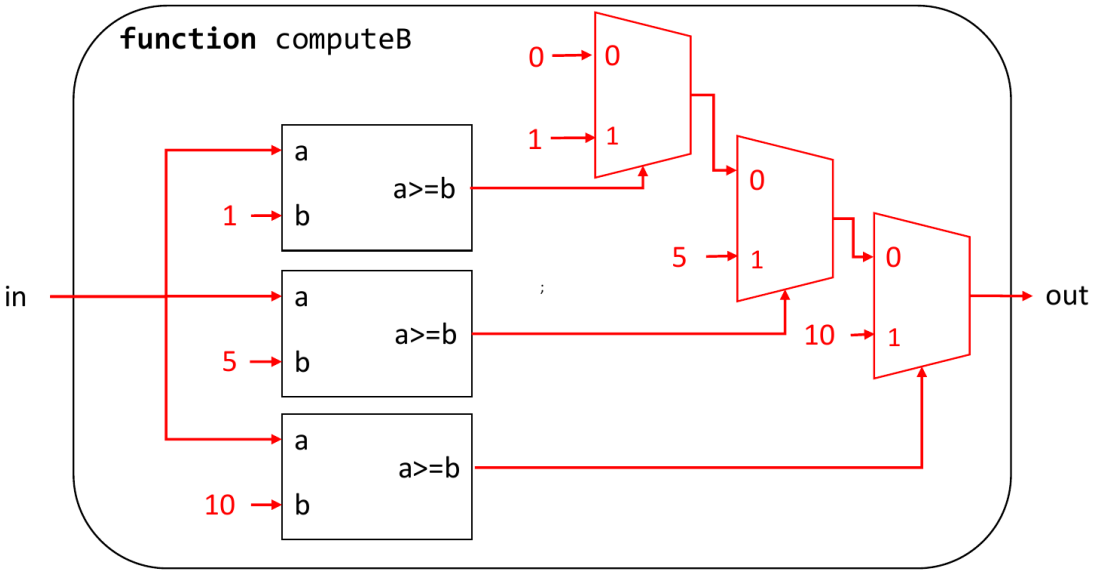
- At the final return statement, we see that we return true if either a = b or a < b.

We see that this logic implements an n-bit comparator. This can be condensed into the single ternary expression that returns 1'b1 if a <= b and 1'b0 otherwise.

The combinational logic here is quite subtle, but this successive comparison structure is fairly common. More details about this kind of comparator can be found in Lab 4.

(D) 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
```



The input must be compared with 1, 5, and 10. We therefore need to supply these as inputs into the compare blocks. The result of the first compare block determines whether out is 1 or 0. This can then be overwritten by the next compare with 5 which can then be overwritten in the compare with 10. This leads to the successive muxes on the output.

Problem 5. From Past Quizzes (Fall 2020)

(A) 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.

x[1]	x[0]	out[1]	out[0]
0	0	**0**	**0**
0	1	**0**	**1**
1	0	**1**	**0**
1	1	**0**	**0**

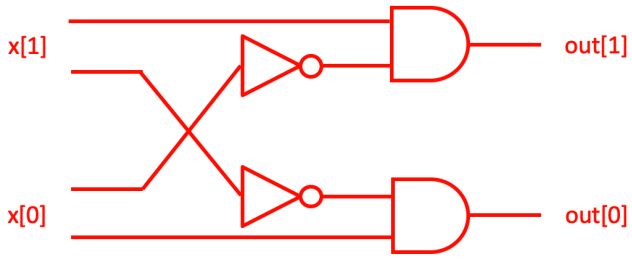
(B) Implement the `mod3` function in Minispec by filling in the function definition below. **Your code should describe a circuit that when synthesized manually without optimizations results in at most 4 (one or two input) logic gates.** Your solution can use the following gates: inverter, 2-input AND,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

```
function Bit#(__2___) mod3 (Bit#(__2___) x);

   return __{x[1] & ~x[0], ~x[1] & x[0] }_____________________;

endfunction
```

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



(D) Complete the truth table for the following Minispec function.

```
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
```

a[2]	a[1]	a[0]	ret[2]	ret[1]	ret[0]
0	0	0	**0**	**0**	**1**
0	0	1	**0**	**0**	**0**
0	1	0	**0**	**1**	**1**
0	1	1	**1**	**0**	**0**
1	0	0	**0**	**0**	**1**
1	0	1	**1**	**0**	**1**
1	1	0	**0**	**0**	**1**
1	1	1	**1**	**0**	**0**

Problem 6: From Past Quizzes (Fall 2020)

For an unsigned *n*-bit number x,

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

Complete the following Minispec function 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}.

```
function __Bit#(2)__ mod3#(__Integer n__)(__Bit#(n)__ x);
    if (n==1) return ____zeroExtend(x);____
    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 (upper = lower + 1) return ____2;____
            else if ((lower == 0) && (upper==2))return ____1;____
            return ____lower-upper;____
        end else begin
            return ____mod3#(3)({0,lower}+{0,upper});____
        end
    end
endfunction
```

The key part of the formula is that we subtract the numbers in the odd indices (0-indexed) and we add the numbers in the even indices.
Suppose h is odd, i.e., we take the first branch. Well, whatever was odd in the original number is also odd in the lower half, and whatever was even in the original number is also even in the lower half. So the result from this recursive call is correct.

Now let's look at the upper half. x[h] was at an odd index in the original number but now it's at an even index in the recursive call (0). Thus, the odds and evens are shifted. To get the right answer, we need to multiply upper by -1. Hence, why we do lower - upper. If lower is 0 and upper is 1, this would yield -1. We're working in mod 3 so -1 + 3 = 2.
In the bottom half, when h is even, nothing is shifted and we can just return lower + upper. We want to return one of 0,1, or 2, so if we have an overflow, (e.g. 2+1 or 2+2), we can add an extra zero so we can represent the sum and take the mod of the sum by calling the function again.

This isn't necessary for the problem, but some more intuition for the formula is that every even power of 2 (2**i when i is even) is 1 greater than a multiple of 3. This is because they're powers of 4, and 4 is 1 mod 3. So taking the mod 3 of 2**i*x[i], we get x[i].And every odd power of 2 is one less than a power of 2. This is because 2 is 2 mod 3, and we're multiplying by 4, which is 1 mod 3. So the product is always 2 mod 3. So taking the mod 3 of that gives us -x[i].