6.004 Tutorial Problems
L05 – Combinational Logic

Problem 1

Create the following logic gates using only NAND gates (truth tables are shown with each problem):
Note: There are multiple potential solutions

A Inverter

NAND(A, A)

A	Out
0	1
1	0

B AND gate (you may also use inverters)

INV(NAND(A, B))

A	B	Out
0	0	0
0	1	0
1	0	0
1	1	1

C OR gate (you may also use AND gates and Inverters)
Hint: Consider DeMorgan's law!

NAND(INV(A), INV(B))

A	**B**	**Out**
0	0	0
0	1	1
1	0	1
1	1	1

D XOR gate (you may also use AND gates, OR gates, and Inverters)

AND(OR(A, B), NAND(A, B))

A	**B**	**Out**
0	0	0
0	1	1
1	0	1
1	1	0

Problem 2 Mini

Consider the 3-input Boolean function G(A,B,C) = .

There's good news and bad news: the bad news is that the stockroom only has G gates. The good news is that it has as many as you need. Using only combinational circuits built from G gates, one can implement (choose the best response):

a Any Boolean function (G is functionally complete)
b Only functions with 3 inputs or less
c Only functions with the same truth table as G

A. Multiple ways to do this. Here are three:
1. Fix B at 1 and you get NOR(A, C),
2. Fix C at 0 and you get NAND(A,B)
NAND and NOR are both functionally complete
3. Instead define AND, OR, and NOT

An Aside on Bluespec

Multi-bit Extraction

We can extract multiple bits from a bit vector using a square bracket [] and an inclusive range (upper_index:lower_index) in Bluespec (e.g., x[32:10]). This operator differs from the Python's one used for indexing arrays. The valid index range is N-1 ~ 0 for a N-bit variable.

Note that multiple bit extracting range can only be indexed starting from a more significant bit (higher index) to a less significant bit (lower index).

- x[5:1] is same as {x[5], x[4], x[3], x[2], x[1]}
- x[2:0] is same as {x[2],x[1],x[0]}
- x[1:1] is same as x[1]
- x[1:3] is illegal
- x[1:5] is illegal

Bluespec Integer Literals

Bluespec expresses integers in several manners. Bluespec Integer Literals could be sized or unsized and could be binary, decimal or hexadecimal.

Sized Literals

1. 5'b10011
2. 8'hAB
3. 12'd10

Unsized Literals

4. '1
5. 'b1100_0110
6. 'hFFCC_0110
7. 'd70

The case statement

```
case ( expression_to_evaluate )
   case_item1, case_item2:  expression1;
   case_item3:              expression3;
   default:             default_expression;
endcase
```

Problem 3

A Using only AND (&), OR (+), and NOT(!), write a 2:1 multiplexer function in Bluespec.

```
function Bit#(1) two_to_one_mux(Bit#(2) in, Bit#(1) sel);
		Bit#(1) ret = (sel & in[1]) | (~sel & in[0]);
		return ret;
endfunction
```

B Write the same function, but use conditionals instead (if, ?:).

```
function Bit#(1) two_to_one_mux(Bit#(2) in, Bit#(1) sel);
		Bit#(1) ret = sel ? in[1] : in[0];
		return ret;
endfunction
```

C Write a 4:1 multiplexer function in Bluespec, using case statements.

```
function Bit#(1) four_to_one_mux(Bit#(4) in, Bit#(2) sel);
	Bit#(1) ret = 0;
	case (sel)
		2'b00: ret = in[0];
		2'b01: ret = in[1];
		2'b10: ret = in[2];
		2'b11: ret = in[3];
	endcase

	return ret;
endfunction
```

Problem 4

Implement a 4-bit ripple carry adder in Bluespec using the half adder (ha) and full adder (fa) functions that we discussed in the lecture. The function specification for this adder should look like:

```
function Bit#(4) addRecitation(Bit#(4) a, Bit#(4) b, Bit#(1) c_in);
```

Note that this function specification assumes that the *final carry out* is ignored.

```
function Bit#(4) addRecitation(Bit#(4) a, Bit#(4) b, Bit#(1) c_in);
   Bit#(2) b0 = fa(a[0], b[0], c_in);
   Bit#(2) b1 = fa(a[1], b[1], b0[1]);
   Bit#(2) b2 = fa(a[2], b[2], b1[1]);
   let b3 = fa(a[3], b[3], b2[1]);     // we can use let

   return {b3[0], b2[0], b1[0], b0[0]};
endfunction
```