6.004 Tutorial Problems
L08 – Combinational Devices and Introduction to Minispec

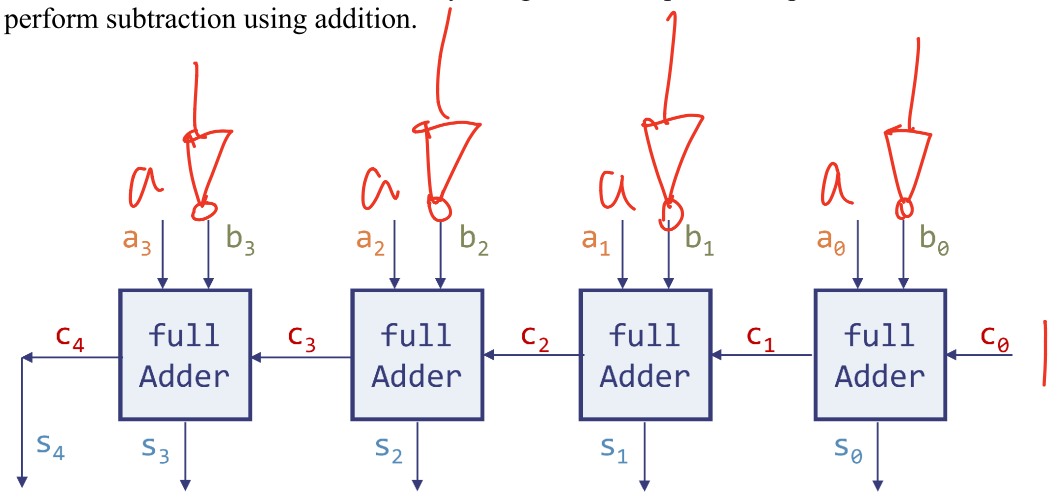
Note: A subset of essential problems are marked with a red star (★). We especially encourage you to try these out before recitation.

Problem 1. ★

(A) Consider the 4-bit ripple carry adder we saw in lecture. Its circuit is shown below. Modify the diagram to build a **subtractor**, i.e., a circuit that given 4-bit inputs *a* and *b*, computes *a* – *b*.

You may use only one ripple-carry adder, and may add at most four gates to the diagram. Assume that *a* and *b* use two's complement representation. Your circuit should return the result in two's complement representation.

Hint: Back in lecture 1, we saw that by using two's complement representation, we could perform subtraction using addition.



(B) Implement your subtractor as a Minispec function `sub4`. Your function can use at most one `rca4` function (the function implementing 4-bit ripple carry adder we saw in lecture).

```
function Bit#(5) sub4(Bit#(4) a, Bit#(4) b);

    return rca4(a, ~b, 1'b1);

endfunction
```

$a - b$

$a + (-b)$

$\boxed{a + (\sim b + 1)}$

1 binary

0b1

Problem 2. ★

X[0] python

(A) Implement a Minispec function `isZero` that returns 1 if its 4-bit input is zero, and 0 otherwise. Your implementation can only use bitwise logical operations and bit selection, and cannot use the equality/inequality operators.

0 0 0 0

X[3] X[2] x[1] x[0]

```
function Bit#(1) isZero(Bit#(4) x);
```

return ~X[3] & ~x[2] & ~x[1] & ~x[0]

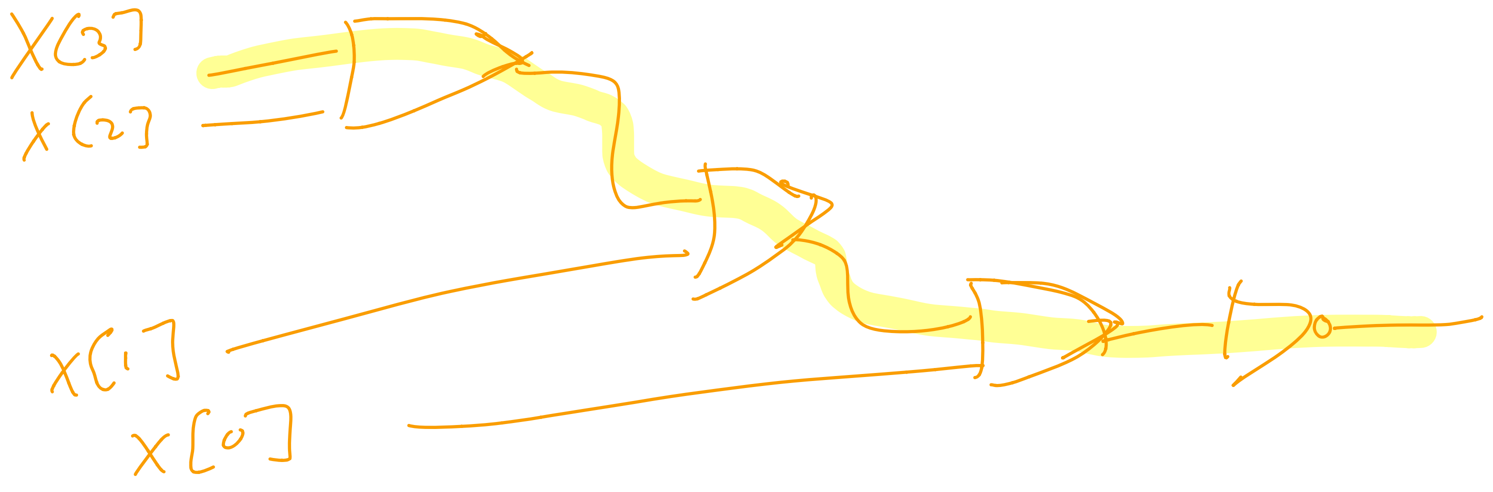
```
endfunction
```

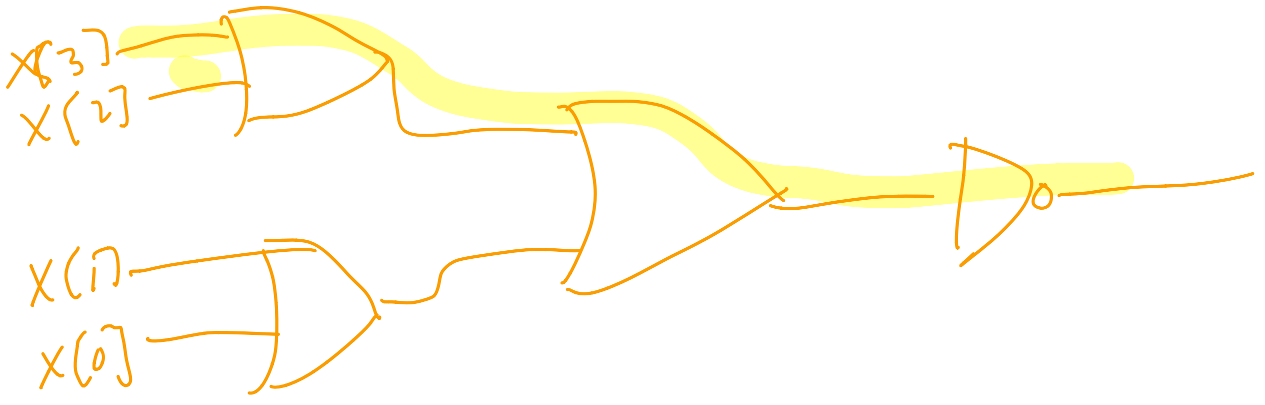
de morgan's

$\overline{x[3] + x[2] + x[1] + x[0]}$

return ~(x[3] | x[2] | x[1] | x[0])

(B) Manually synthesize your function into a combinational circuit using 2-input AND gates, 2-input OR gates, and inverters. Keep delay low by minimizing the number of logic gates between input and output. Draw the resulting circuit.





= 0x8

0x 0 0 0 8

Problem 3. ★

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#(4) upper = 4'hB; // hex value 0xB
   Bit#(4) lower = (c == 1)? 4'h8 : 4'h7;
   Bit#(8) x = {upper, lower};
   Bit#(2) ret = case ({a,b})
         0: 1;
         1: x[1:0];
         2: x[3:2];
         3: x[7:6] ^ 2'b11;
   endcase;
   return ret;
endfunction
```

if (c==1)
  return 4'h8
else :
  return 4'h7

a = 0b101
b = 0b010

{a,b}
= 0b101010

if {a,b}==0 ;
  return 1
else if {a,b}==1 :
  return x[1:0]
else if {a,b}==2 :
  return x[3:2]
else if {a,b}==3 :
  return x[7:6]^2'b11

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		

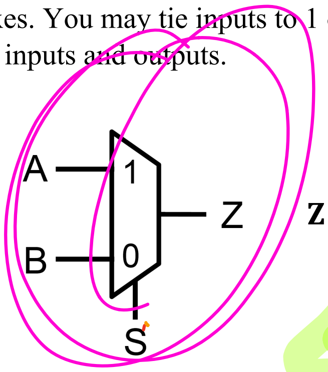
NAND $\overline{ab} = \overline{a} + \overline{b}$

$\underbrace{\overline{ab}}_{\text{not, and}} = \underbrace{\overline{a} + \overline{b}}_{\text{not, or}}$

Problem 4.

Show that 1-bit 2-to-1 muxes are universal, i.e., they can be used to implement any combinational circuit. To show universality, implementing an inverter, an AND gate, and an OR gate using only 1-bit 2-to-1 muxes. You may tie inputs to 1 or 0 if necessary, and may use one or multiple muxes. Clearly label all inputs and outputs.

if S==1
return A
else if S==0:
return B

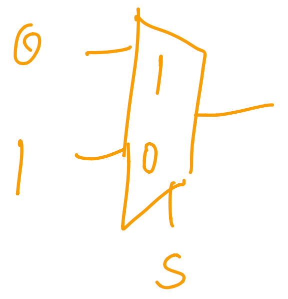


$\mathbf{Z = A \cdot S + B \cdot \overline{S}}$

$Z(A, B, S)$

NOT

Logic diagram of inverter implementation using 2-input mux:



$Z = \overline{S}$

$Z(0, 1, S)$

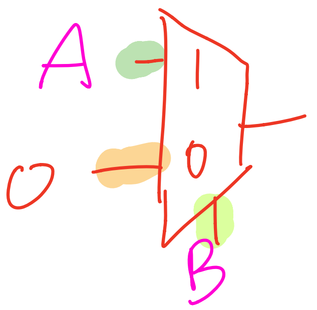
Logic diagram of AND gate implementation using 2-input mux:

NAND

$Z = A \cdot S + \cancel{B \cdot \overline{S}}$

0

NOT(AND(A,B))



$Z = A \cdot S$

$Z(A, 0, B)$

NOT($Z(A, 0, B)$)

Logic diagram of OR gate implementation using 2-input mux:

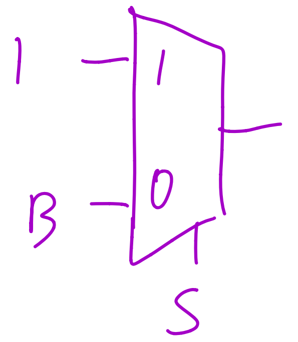
$Z = A \cdot S + B \cdot \overline{S}$

$Z(1, B, S)$

set A=1 $= S + B \cdot \overline{S}$

$= S + B$

A OR B

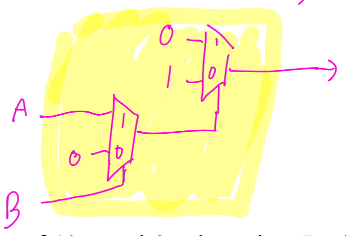


$Z = S + B$

$Z(1, A, B)$

$\overline{a} + \overline{b}$

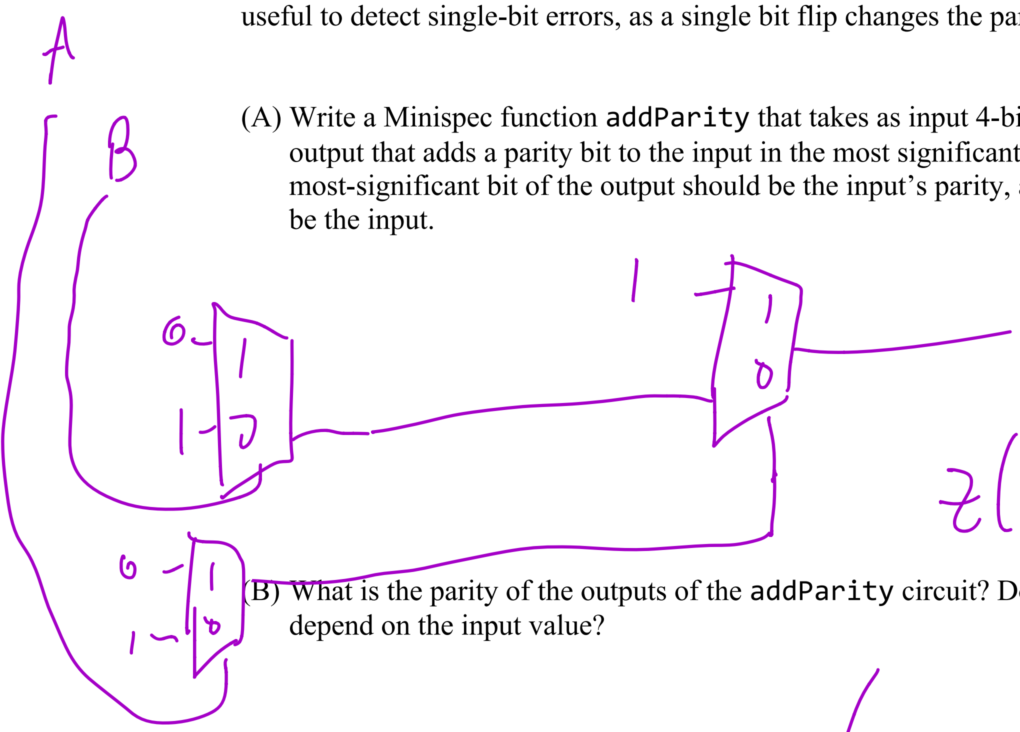
$NAND(A,B) = z(0,1,z(A,0,B))$



Problem 5.

The parity of an n-bit number *x* is 1 if *x* has an odd number of 1's, and 0 otherwise. Parity is useful to detect single-bit errors, as a single bit flip changes the parity of a value.

(A) Write a Minispec function `addParity` that takes as input 4-bit value and returns a 5-bit output that adds a parity bit to the input in the most significant position. In other words, the most-significant bit of the output should be the input's parity, and the remaining bits should be the input.



NAND

$\bar{a} + \bar{b} = \overline{ab}$

$z(1, \quad , \quad)$

$z(0,1,A)$ $z(0,1,B)$

(B) What is the parity of the outputs of the `addParity` circuit? Does the parity of the output depend on the input value?

(C) Write a Minispec function `checkParity` that takes as input a 5-bit value and returns True if the input has an even number of 1's, and returns False otherwise.

$z(1, z(0,1,A), z(0,1,B))$

Problem 6. Combinational Minispec (part of Spring 2020 Quiz 2 problem 3, 8 points)

Complete the truth table for the following Minispec function.

```
function Bit#(3) h(Bit#(1) a, Bit#(2) b);
    Bit#(3) ret = 3'b110;
    case ({a, b[1]})
        0: ret = {1'b0, zeroExtend(a) & b};
        1: ret = zeroExtend(a) + signExtend(b);
        3: ret = {a, ~b};
        default: ret = 3'b010;
    endcase
    return ret;
endfunction
```

a	b[1]	b[0]	ret[2]	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			

(Label: 3A) Copy the truth table and fill in all the missing blanks.