6.004 Tutorial Problems
L08 – Combinational Devices and Introduction to Minispec

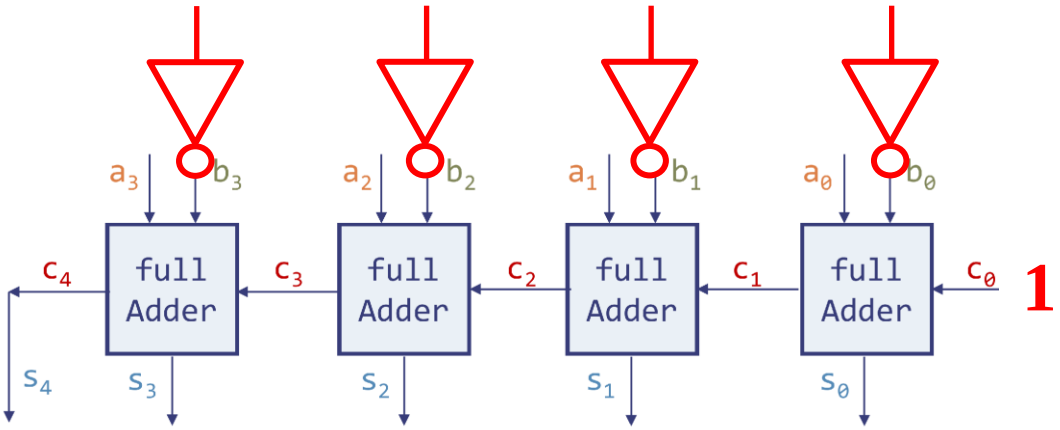
Note: A subset of essential problems are marked with a red star (★). We especially encourage you to try these out before recitation.

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.



We use the trick we saw in lecture 1, that −b = ~b + 1: negating a number is the same as inverting all bits and adding 1. Adding NOT gates to each b input inverts all bits, while wiring the rightmost carry-in to 1 (instead of 0) adds 1 to the entire sum.

(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
```

Exactly the trick described above. Note that in Minispec, writing a tilde ~ before an expression inverts all bits of the result, also called taking the complement. In the final constant `1'b1` the

first part `1'b` indicates that the constant is 1 bit wide and the following digits are really in binary (represented by `b`). Only the final `1` actually indicates that the constant is the bit 1. This is a very explicit way to write this constant; we could also have just written 1 here. Sometimes this explicit notation is easier though, for example if we wanted an 8 bit number where all but the second bit from the left as 1, 8'b10111111 does the job and is quicker than figuring out the hex or decimal value that we want.

Problem 2.

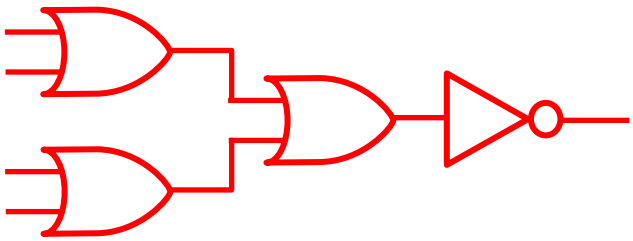
(A) Implement a Minispec function `isZero` that returns 1 if its 4-bit input is zero, and 0 otherwise. Your implementation can only use bitwise logical operations and bit selection, and cannot use the equality/inequality operators.

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

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

endfunction
```

This Minispec expression means "NOT (x[3] OR x[2] OR x[1] OR x[0])". As before, the tilde ~ inverts all bits of the expression, in this case just a single bit; and the vertical bar | is binary bitwise OR.

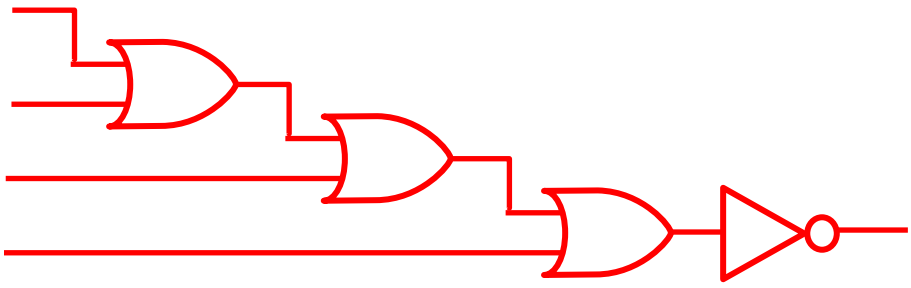
(B) Manually synthesize your function into a combinational circuit using 2-input AND gates, 2-input OR gates, and inverters. Keep delay low by minimizing the number of logic gates between input and output. Draw the resulting circuit.

We draw three OR gates in a small tree and a NOT gate at the end. Because OR is associative, any tree of ORs that takes all four inputs will be logically correct, but this balanced tree arrangement minimizes the delay.



This is also a valid synthesis of isZero, but it has worse delay; the top two inputs have to pass through all three OR gates and the NOT gate to reach the output. So it would **not** satisfy the problem statement.

BAD:



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
```

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

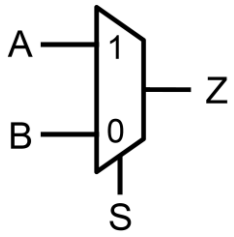
- In the first case, `{a, b} = 0 = 0b00` if `a = 0` and `b = 0`, which gives us the first two rows of the table. `ret` is 1 and is a 2-bit number, so `ret = 0b01`, so `ret[1] = 0` and `ret[0] = 1`, regardless of what `c` is. (Remember that `ret[0]` is the rightmost bit and higher indices give more significant bits.)
- In all the later cases, we need to compute the value of `x`. We have `x = {upper, lower}`, the concatenation of `upper` and `lower`. `upper` is the 4-bit number `0xB = 0b1011`, and if `c = 0` then `lower` is the 4-bit number `8 = 0b1000`, but if `c = 1` then `l` is the 4-bit number `7 = 0b0111`. So:
  - If c = 0 then `x = {upper, lower} = 0b10110111`
  - If c = 1 then `x = {upper, lower} = 0b10111000`

  So now:
- In the second case, `{a, b} = 1 = 0b01` if `a = 0` and `b = 1`.
  - If `c = 0` then we want the last two bits of `0b10110111`, which are `0b11`.
    - So in row 3, `ret[1] = 1` and `ret[0] = 1`.
  - If `c = 1` then we want the last two bits of `0b10111000`, which are `0b00`.

        - So in row 4, `ret[1] = 0` and `ret[0] = 0`.
- In the third case, `{a, b} = 2 = 0b10` if `a = 1` and `b = 0`.
    - If `c = 0` then we want the fourth-to-last and third-to-last bits of `0b10110111`, which are `0b01`.
        - So in row 5, `ret[1] = 0` and `ret[0] = 1`.
    - If `c = 1` then we want the fourth-to-last and third-to-last bits of `0b10111000`, which are `0b10`.
        - So in row 6, `ret[1] = 1` and `ret[0] = 0`.
- In the fourth case, `{a, b} = 3 = 0b11` if `a = 1` and `b = 1`.
    - If `c = 0` then we want the leftmost two bits of `0b10110111`, which are `0b10`, XORed with `0b11`. This is equal to `0b01`.
        - So in row 7, `ret[1] = 0` and `ret[0] = 1`.
    - If `c = 1` then we want the leftmost two bits of `0b10111000`, which are `0b10`, XORed with `0b11`. This is equal to `0b01`.
        - So in row 8, `ret[1] = 0` and `ret[0] = 1`.

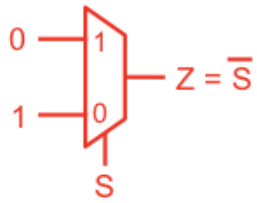
Problem 4.

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.



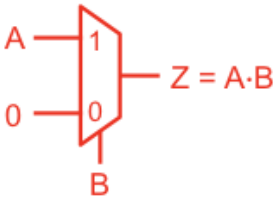
$$\mathbf{Z = A \cdot S + B \cdot \bar{S}}$$

Logic diagram of inverter implementation using 2-input mux:



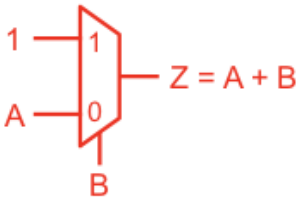
If S is 0, the mux selects and outputs the bottom input to the part labeled 0, which is 1. If S is 1, the mux selects and outputs the top input to the part labeled 1, which is 0.

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



If B is 0, the mux selects and outputs the bottom input to the part labeled 0, which is 0. This makes sense because (0 AND anything) is 0. If B is 1, the mux selects and outputs the top input to the part labeled 1, which is A. Indeed, (1 AND any input) is that input itself.

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



If B is 0, the mux selects and outputs the bottom input to the part labeled 0, which is A. This makes sense because (0 OR any input) is that input itself. If B is 1, the mux selects and outputs the top input to the part labeled 1, which is 1. Indeed, (1 AND anything) is 1.

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.

```
function Bit#(5) addParity(Bit#(4) in);
    Bit#(1) parity = in[3] ^ in[2] ^ in[1] ^ in[0];
    return {parity, in};
endfunction
```

If you XOR many bits together, the result is exactly the parity of those bits. One way to see this is to observe that both operations are equivalent to addition mod 2.

After computing the parity bit, we make it the most significant bit of the result, concatenated with the original input, with the `{ , }` concatenation operator in Minispec.

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

Always 0, independently of the input, because the output always has an even number of 1's. If the input had an odd number of 1's, the parity bit is 1, making an even total number of 1's. If the input had an even number of 1's, the parity bit is 0, so the total number of 1's stays even.

(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.

```
function Bool checkParity(Bit#(5) in);
    Bit#(1) parity = in[4] ^ in[3] ^ in[2] ^ in[1] ^ in[0];
    return (parity == 0);
endfunction
```

Remember that 0 and 1 are not the same as False and True in Minispec. 0 and 1 have type Bit#(1) (or Bit#(n) for other n), and False and True have type Bool.

Minispec comparison operators like `==` return values of type Bool, so if we wanted to get a Bool out of some Bit#(1)s, we could use comparison operators, as seen above. (If we wanted to go the other way, to get Bit#(1) out of a Bool, we could use the ternary expression `( ? : )`. The syntax is `(condition ? value_if_true : value_if_false)`.)