COMS10015 lab. worksheet #1

§1. C-class, or core questions

▷ **S1[C].** This question is more like a limited form of guided explanation (or tutorial); as such, there is no associated solution.

▷ **S2[C].** a The first three cases should all be easy to replicate because a) they only need a few NAND operators, and b) were covered in the lecture slot(s). Specifically, we saw the following identities:

$$\begin{array}{lcl} \neg x & \equiv & x \mathbin{\overline{\wedge}} x \\ x \wedge y & \equiv & (x \mathbin{\overline{\wedge}} y) \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} y) \\ x \vee y & \equiv & (x \mathbin{\overline{\wedge}} x) \mathbin{\overline{\wedge}} (y \mathbin{\overline{\wedge}} y) \end{array}$$

Note that to compute $x \mathbin{\overline{\wedge}} x$, you need two jumper wires connected to the NAND operator input pins; both need either to be disconnected (meaning $x = 0$) or connected to 1, because both represent the same x. This is is a little awkward, so it might be easier to use the fact that

$$\neg x \equiv x \mathbin{\overline{\wedge}} 1.$$

Now the second input pin is fixed to 1, meaning you only have one jumper wire representing x (more closely matching the NOT operator).

The case of XOR is more difficult. We saw in the lecture slot(s) that it can be written in various different ways, e.g.,

$$\begin{array}{rcl} x \oplus y & \equiv & (\neg x \wedge y) \vee (x \wedge \neg y) \\ & \equiv & (x \vee y) \wedge \neg (x \wedge y) \end{array}$$

For the sake of argument imagine we instead opt for the former: just by applying the identities above, we can rewrite it as

$$\begin{array}{lcll} t_0 & = & x & \overline{\wedge}\ x \\ t_1 & = & y & \overline{\wedge}\ y \\ t_2 & = & t_0 & \overline{\wedge}\ y \\ t_3 & = & t_2 & \overline{\wedge}\ t_2 \\ t_4 & = & t_1 & \overline{\wedge}\ x \\ t_5 & = & t_4 & \overline{\wedge}\ t_4 \\ t_6 & = & t_3 & \overline{\wedge}\ t_3 \\ t_7 & = & t_5 & \overline{\wedge}\ t_5 \\ t_8 & = & t_6 & \overline{\wedge}\ t_7 \end{array}$$

so that $x \oplus y \equiv t_8$ having used 9 operators. We can then make incremental improvements by inspection. t_3 computes $\neg t_2 = t_2 \mathbin{\overline{\wedge}} t_2$ for example, and then later t_6 computes $\neg t_3$ in the same way. So basically, since $t_6 = \neg\neg t_2$, we can eliminate both NOTs via the involution axiom, and similarly for t_7, to get

$$\begin{array}{lcll} t_0 & = & x & \overline{\wedge}\ x \\ t_1 & = & y & \overline{\wedge}\ y \\ t_2 & = & t_0 & \overline{\wedge}\ y \\ t_4 & = & t_1 & \overline{\wedge}\ x \\ t_8 & = & t_2 & \overline{\wedge}\ t_4 \end{array}$$

having now used only 5 operators: this *seems* to be about the best we can hope for. But imagine we opt for the latter expression that describes XOR instead of the former. Some initial manipulation allows us to rewrite it as follows

$$\begin{array}{rcll} x \oplus y & \equiv & (x \vee y) \wedge \neg (x \wedge y) & \\ & \equiv & \neg (x \wedge y) \wedge (x \vee y) & \text{(commutativity)} \\ & \equiv & (x \wedge \neg (x \wedge y)) \vee (y \wedge \neg (x \wedge y)) & \text{(distribution)} \\ & \equiv & \neg\neg ((x \wedge \neg (x \wedge y)) \vee (y \wedge \neg (x \wedge y))) & \text{(involution)} \\ & \equiv & \neg (\neg (x \wedge \neg (x \wedge y)) \wedge \neg (y \wedge \neg (x \wedge y))) & \text{(NAND)} \\ & \equiv & (x \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} y)) \mathbin{\overline{\wedge}} (y \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} y)) & \end{array}$$

It might seem odd to call this simplification, because the result may look more complicated! Crucially however, *every* (sub-)term is now computed using NAND: the expression can therefore be rewritten as

$$\begin{array}{lcll} t_0 & = & x & \overline{\wedge}\ y \\ t_1 & = & x & \overline{\wedge}\ t_0 \\ t_2 & = & y & \overline{\wedge}\ t_0 \\ t_3 & = & t_1 & \overline{\wedge}\ t_2 \end{array}$$

so that $x \oplus y \equiv t_3$ now using just 4 operators.

Given an expression in SoP (resp. PoS) form, the example above yields a fairly general strategy for simplification ready for implementation using NAND (resp. NOR) alone. In short, introducing what *seem* to be redundant NOT operators turns out to be an advantage, because we can then apply the de Morgan axiom: this "pushes" a NOT into the expression, swapping ANDs into NANDs etc.

b The first step is to formulate a working design. We want to produce the following behaviour, as translated from the description:

a	b	c	$\text{MAJ}(a, b, c)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Note that if three of the inputs are 1 then obviously two of them are 1, so it suffices to detect the latter case only. This reduces the amount of work required, because we already know AND will produce 1 as an output if both inputs are 1: we can use AND to "detect" each of three cases where two or more inputs are 1. Put another way, we can express the function as

$$\text{MAJ}(a, b, c) = (b \wedge c) \vee (a \wedge c) \vee (a \wedge b),$$

i.e., "the output is 1 when either $b = 1$ and $c = 1$, or $a = 1$ and $c = 1$, or $a = 1$ and $b = 1$".

The next step is to translate this into NAND operators. Conceptually this is easy, because we just apply the known identities (as we did above with XOR). However, this quickly becomes hard to manage: when written out naively in full, we use around 39 NAND operators. With more care, we can share numerous common sub-terms and write

$$\begin{array}{lcl}
t_0 &=& b \mathbin{\overline{\wedge}} c \\
t_1 &=& a \mathbin{\overline{\wedge}} c \\
t_2 &=& a \mathbin{\overline{\wedge}} b \\
t_3 &=& t_0 \mathbin{\overline{\wedge}} t_0 \\
t_4 &=& t_1 \mathbin{\overline{\wedge}} t_1 \\
t_5 &=& t_2 \mathbin{\overline{\wedge}} t_2 \\
t_6 &=& t_3 \mathbin{\overline{\wedge}} t_3 \\
t_7 &=& t_4 \mathbin{\overline{\wedge}} t_4 \\
t_8 &=& t_5 \mathbin{\overline{\wedge}} t_5 \\
t_9 &=& t_6 \mathbin{\overline{\wedge}} t_7 \\
t_{10} &=& t_9 \mathbin{\overline{\wedge}} t_9 \\
t_{11} &=& t_8 \mathbin{\overline{\wedge}} t_{10}
\end{array}$$

where $\text{MAJ}(a, c, b) = t_{11}$ having used 12 operators. But we can again make improvements to this with initial simplification of our expression. Applying roughly the same strategy as with XOR, we find that

$$\begin{array}{rcll}
\text{MAJ}(a, b, c) &=& (b \wedge c) \vee (a \wedge c) \vee (a \wedge b) & \\
&\equiv& (b \wedge c) \vee (a \wedge (c \vee b)) & \text{(distribution)} \\
&\equiv& \neg\neg((b \wedge c) \vee (a \wedge (c \vee b))) & \text{(involution)} \\
&\equiv& \neg(\neg(b \wedge c) \wedge \neg(a \wedge (c \vee b))) & \text{(de Morgan)} \\
&\equiv& \neg(\neg(b \wedge c) \wedge \neg(a \wedge \neg\neg(c \vee b))) & \text{(involution)} \\
&\equiv& \neg(\neg(b \wedge c) \wedge \neg(a \wedge \neg(\neg c \wedge \neg b))) & \text{(NAND)} \\
&\equiv& (b \mathbin{\overline{\wedge}} c) \mathbin{\overline{\wedge}} (a \mathbin{\overline{\wedge}} ((c \mathbin{\overline{\wedge}} c) \mathbin{\overline{\wedge}} (b \mathbin{\overline{\wedge}} b))) &
\end{array}$$

and produce an implementation

$$\begin{array}{lcl}
t_0 &=& b \mathbin{\overline{\wedge}} c \\
t_1 &=& c \mathbin{\overline{\wedge}} c \\
t_2 &=& b \mathbin{\overline{\wedge}} b \\
t_3 &=& t_1 \mathbin{\overline{\wedge}} t_2 \\
t_4 &=& a \mathbin{\overline{\wedge}} t_3 \\
t_5 &=& t_0 \mathbin{\overline{\wedge}} t_4
\end{array}$$

where $\text{MAJ}(a, c, b) = t_5$ now using just 6 operators.

§2. R-class, or revision questions

▷ **S3[R].** There is a set of solutions available at

https://assets.phoo.org/COMS10015_2025_TB-4/csdsp/sheet/misc-revision_s.pdf

§3. A-class, or additional questions

▷ **S4[A].** To start with, think a bit about what the majority function is doing: you can think of it as an exercise in voting, with the output representing a majority decision between voters a, b, and c. For instance if two or more vote 1 then the output is 1; otherwise two or more must have voted 0 meaning the output is 0.

By using the majority function as suggested in the question, you produce a component called a **C-element**. Now there are only two voters, but also a "default result" that decides the output in case of a tie. Put another way, we can write the as

a	b	$c' = \text{MAJ}(a, b, c)$
0	0	0
0	1	c
1	0	c
1	1	1

where c and c' are the values of c before and after we set a and b. The middle two rows are saying that if the value was $c = 0$ (resp. $c = 1$), then it will *remain* $c' = 0$ (resp. $c' = 1$); in contrast, the top and bottom rows are saying that c' is *forced* to 0 and 1 respectively, regardless of what c was before.

In short, this component retains or remembers some state over time; this differs significantly from those previously seen, the majority function for example, where the output changes (and is lost) as soon as the inputs change. Since c is representing 1 bit, the C-element acts as a cell that can store 1-bit values, very roughly like a memory cell does. In later lecture slot(s), we see that it falls within a larger class of components called latches.