COMS10015 lab. worksheet #3

Although some questions have a written solution below, for others it will be more useful to experiment in a hands-on manner (e.g., using a concrete implementation). The file

https://assets.phoo.org/COMS10015_2024_TB-4/csdsp/sheet/lab-03_s.tar.gz

supports such cases.

§1. C-class, or core questions

▷ **S1[C].** There is no associated solution for this question, because it is essentially a guided explanation rather than a task or question per se.

▷ **S2[C].** An associated solution, i.e., a (documented) implementation, for this question can be found in the archive provided. Note that one can produce such a solution by taking content from the lecture slot(s), then translating (or "porting") it into a LogisimEvo implementation: for reference, Figure 1 captures that content.

▷ **S3[C].** An associated solution, i.e., a (documented) implementation, for this question can be found in the archive provided.

NOT, AND, and OR should all be fairly straightforward to replicate, because the following identities were introduced in the lecture slot(s):

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

For each case, a Logisim implementation is reproduced within the archive provided. XOR is more difficult, however. We saw in the lecture slots(s) that it can can be expressed 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 expression: by applying the identities above, we can rewrite it as

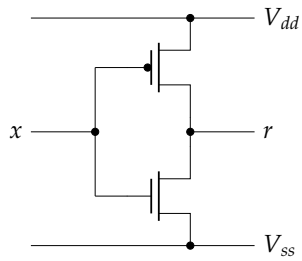
$$\begin{array}{rcl} t_0 & = & x \mathbin{\overline{\wedge}} x \\ t_1 & = & y \mathbin{\overline{\wedge}} y \\ t_2 & = & t_0 \mathbin{\overline{\wedge}} y \\ t_3 & = & t_2 \mathbin{\overline{\wedge}} t_2 \\ t_4 & = & t_1 \mathbin{\overline{\wedge}} x \\ t_5 & = & t_4 \mathbin{\overline{\wedge}} t_4 \\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

so that $x \oplus y \equiv t_8$ using 9 operators. We can then try to realise incremental improvements by inspection. Notice that $t_3 = t_2 \mathbin{\overline{\wedge}} t_2 = \neg t_2$ and $t_6 = t_3 \mathbin{\overline{\wedge}} t_3 = \neg t_3$, so, since $t_6 = \neg\neg t_2$, we can eliminate both NOTs by applying the involution axiom; a similar fact is true wrt. t_7. As a result, we end up with

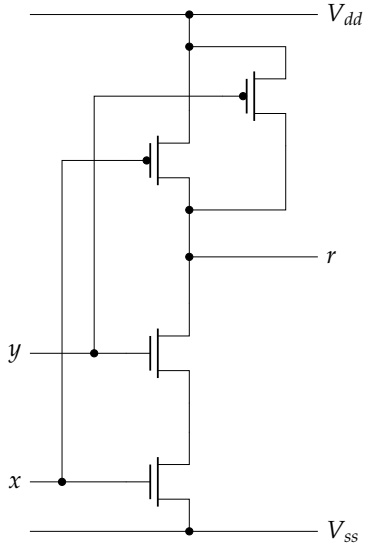
$$\begin{array}{rcl} t_0 & = & x \mathbin{\overline{\wedge}} x \\ t_1 & = & y \mathbin{\overline{\wedge}} y \\ t_2 & = & t_0 \mathbin{\overline{\wedge}} y \\ t_4 & = & t_1 \mathbin{\overline{\wedge}} x \\ t_8 & = & t_2 \mathbin{\overline{\wedge}} t_4 \end{array}$$

so that $x \oplus y \equiv t_8$ now using only 5 operators. However, imagine we opt for the *latter* expression for XOR instead of the *former*. Careful selection of axioms allows us to rewrite it as follows

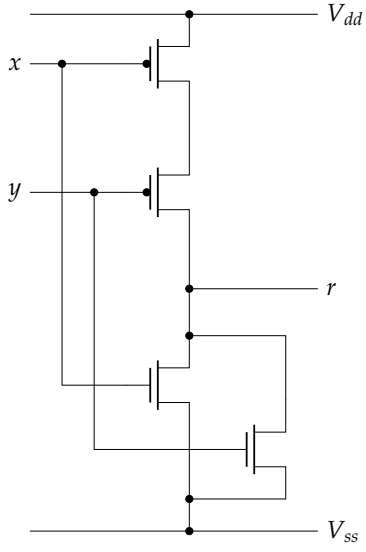
$$\begin{array}{rcll} x \oplus y & \equiv & (x \vee y) \wedge \neg (x \wedge y) & \\ & \equiv & \neg (x \wedge y) \wedge (x \vee y) & \text{(commutativity)} \\ & \equiv & (x \wedge \neg (x \wedge y)) \vee (y \wedge \neg (x \wedge y)) & \text{(distribution)} \\ & \equiv & \neg\neg ((x \wedge \neg (x \wedge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



(a) *A CMOS-based NOT gate.*

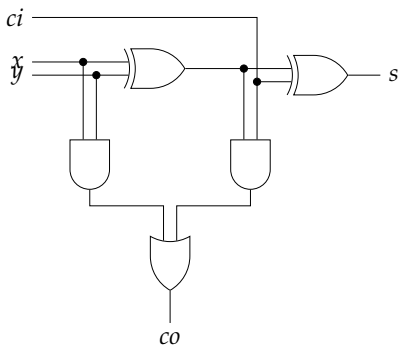


(b) *A CMOS-based,* 2*-input NAND gate.*

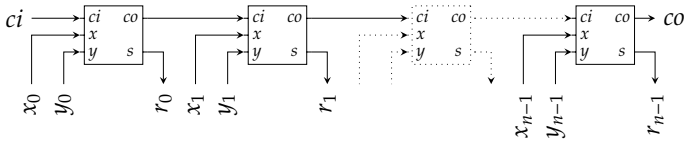


(c) *A CMOS-based,* 2*-input NOR gate.*

Figure 1: *MOSFET-based implementations of NOT, NAND and NOR logic gates.*



(a) *A full-adder cell.*



(b) *A ripple-carry adder.*

Figure 2: *Implementation of a* 1*-bit full-adder cell, and* n*-bit ripple-carry adder based on it.*

It might seem odd to call this simplification, because the result might look *more* complicated! Crucially, however, *every* (sub-)term is being computed using NAND: the expression can therefore be rewritten directly 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 only 4 operators.

Given some expression in SoP (resp. PoS) form, the example above suggests a general strategy where the goal is to produce an implementation using NAND (resp. NOR) alone. In short, introducing what *seem* to be redundant NOT operators turns out to be an advantage, because it allows application of the de Morgan axiom: this "pushes" a NOT into the expression, swapping ANDs into NANDs etc.

▷ **S4[C].** An associated solution, i.e., a (documented) implementation, for this question can be found in the archive provided. Note that one can produce such a solution by taking content from the lecture slot(s), then translating (or "porting") it into a LogisimEvo implementation: for reference, Figure 2 captures that content.

§2. R-class, or revision questions

▷ **S5[R].** There is a set of solutions available at

https://assets.phoo.org/COMS10015_2024_TB-4/csdsp/sheet/misc-revision_s.pdf

§3. A-class, or additional questions

▷ **S8[A].** a Recalling that NAND means "NOT-AND", a natural translation of

$$x \overline{\wedge}\ y \equiv \neg(x \wedge y)$$

to accomodate a third input would be

$$\neg(x \wedge y \wedge z).$$

We can write the required truth table as follows:

x	y	z	$x \wedge y \wedge z$	$\neg(x \wedge y \wedge z)$
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	0

b If you think about AND to start with, we can see that

x	y	z	$x \wedge y \wedge z$	$t = x \wedge y$	$t \wedge z$
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	0	0
1	1	0	0	1	0
1	1	1	1	1	1

and hence say

$$x \wedge y \wedge z \equiv (x \wedge y) \wedge z.$$

Put another way, for AND we *can* realise the 3-input version using two 2-input versions. Replacing the AND gates with NAND gates and given

x	y	$x \overline{\wedge} y$
0	0	1
0	1	1
1	0	1
1	1	0

we can see that

x	y	z	$\neg(x \wedge y \wedge z)$	$t = x \overline{\wedge} y$	$t \overline{\wedge} z$
0	0	0	1	1	1
0	0	1	1	1	0
0	1	0	1	1	1
0	1	1	1	1	0
1	0	0	1	1	1
1	0	1	1	1	0
1	1	0	1	0	1
1	1	1	0	0	1

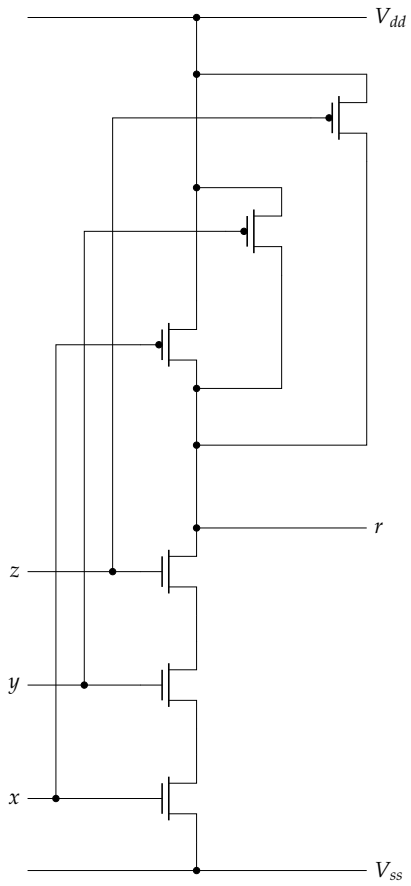
So here, the same fact does not hold, i.e.,

$$\neg(x \wedge y \wedge z) \not\equiv (x \overline{\wedge} y) \overline{\wedge} z,$$

and so we cannot realise a 3-input NAND simply by using 2-input NANDs. Note that parenthesising the expression the other way means inspecting $x \overline{\wedge} (y \overline{\wedge} z)$, but gives a similar result. In both cases, the issue is that we have axioms for AND and OR, but not for NAND and NOR: the reason either fails to produce the result we require is basically because NAND is not associative whereas AND is.

A fast(er) way to answer this question is simply to come up with a counter-example where the two differ (which implies they cannot be equivalent); this relates somewhat to the use of SAT. So for example, stating that for $x = 0$, $y = 0$ and $z = 1$ the 3-input NAND produces 1 whereas the combination of 2-input NANDs produces 0 is more or less enough.

c This question is easier than it sounds; basically we just add extra transistors (one P-MOSFET and one N-MOSFET) to implement a similar approach (highlighted in the next question). Diagrammatically, the result is as follows:



d If you generalise the strategy used for the 2- and 3-input NAND gates, the basic idea is that we need

- a pull-up network of n P-MOSFET transistors that all operate in *parallel* (so if *any* is connected, the result is 1 due to the connection with V_{dd}), and
- a pull-down network of n N-MOSFET transistors that all operate in *series* (so if *all* are connected, the result is 0 due to the connection with V_{ss}).

▷ **S9[A].** An associated solution, i.e., a (documented) implementation, for this question can be found in the archive provided.