- Remember to register your attendance using the UoB Check-In app. Either
  1. download, install, and use the native app[a] available for Android and iOS, or
  2. directly use the web-based app available at

     https://check-in.bristol.ac.uk

  noting the latter is also linked to via the `Attendance` menu item on the left-hand side of the Blackboard-based unit web-site.
- The hardware *and* software resources located in the MVB Linux lab(s). (e.g., MVB-1.15 or MVB-2.11) are managed by the Faculty IT Support Team, a subset of IT Services. If you encounter a problem (e.g., a workstation that fails to boot, an error when you try to use some software, or you just cannot log into your account), they can help: you can contact them, to report then resolve said problem, via

  https://www.bristol.ac.uk/it-support

- The lab. worksheet is written *assuming* you work in the lab. using UoB-managed and thus *supported* equipment. If you need or prefer to use your own equipment, however, various *unsupported*[b] alternatives available: for example, *you* could 1) manually install any software dependencies yourself, *or* 2) use the unit-specific Vagrant[c] box by following instructions at

  https://cs-uob.github.io/COMS10015/vm

- The questions are roughly classified as either C (for core questions, that *should* be attempted within the lab. slot), A (for additional questions, that *could* be attempted within the lab. slot), or R (for revision questions). Keep in mind that we only *expect* you to attempt the C-class questions: the other classes are provided *purely* for your benefit and/or interest, so there is no problem with nor penalty for totally ignoring them.
- There is an associated set of solutions is available, at least for the C-class questions. These solutions are there for you to learn from (e.g., to provide an explanation or hint, or illustrate a range of different solutions and/or trade-offs), rather than (purely) to judge your solution against; they often present *a* solution vs. *the* solution, meaning there might be many valid approaches to and solutions for a question.
- Keep in mind that various mechanisms exist to get support with and/or feedback on your work; these include both in-person (e.g., the lab. slot itself) *and* online (e.g., the unit forum, accessible via the unit web-site) instances.

[a] https://www.bristol.ac.uk/students/support/it/software-and-online-resources/registering-attendance
[b] The implication here is that such alternatives are provided in a best-effort attempt to help you: they are experimental, and so *no* guarantees about nor support for their use will be offered.
[c] https://www.vagrantup.com

COMS10015 lab. worksheet #1

§1. C-class, or core questions

▷ **Q1[C].** In the lecture slot(s), a very obvious effort was made to differentiate this unit from other units which also cover Boolean algebra. Despite some overlap at face value, our claimed focus is the practical application versus purely theoretical study of this topic. *If* you needed further convincing, this question should provide it.

An important theoretical result we encountered was the universality of the NAND operator: *all* Boolean operators, and therefore *all* Boolean functions, can be realised by using NAND operators alone. We have developed a kit to allow translation of this result from theory into practice:

A boxed NANDboard kit[a] constitutes

- 1 NANDboard (revision F),
- 1 type-A to type-C USB cable, and
- a set of jumper wires (which can be carefully stripped apart from each other).

Although we cannot let you take the kit away, there should be enough kits for one per student; at the start (resp. end) of each lab. slot that uses it, collect a kit from (resp. return the kit to) a lab. demonstrator. Try to pack the content neatly before return: this will help whoever uses it next, in the sense that they can start work as quickly as possible. There is a chance the kit you collect is either incomplete or defective. Such an event *should* be rare of course, but *if* you identify a problem then let a lab. demonstrator know (e.g., versus just returning it as is): this allows us to solve the problem, or just provide a replacement.

[a] We are very interested in both positive and negative feedback about the kit. If, for instance, you do something interesting or "off piste" with it, we want to hear: take a photograph and drop us an email, for example, or tweet and mention @BristolCS so we can pick it up!

Q1–§1 An overview of the NANDboard

Figure 1 captures the main features of the board in high-level block diagram. When oriented to match and viewed left-to-right, you should be able to identify

a the input group,

b the NAND groups,

c the output group and

d the power group.

Figure 2 offers a more detailed specification of each group (i.e., the internal components and structure), which is explained by the following Sections.

Q1–§2 An overview of the NANDboard: the power group

The power group houses a USB connector: connecting it, via the supplied cable, to the USB port of a host workstation provides a power supply to the board.

Note that above and below the USB connector there are 2 groups of 4 pins. These allow multiple NANDboards to be combined together, but doing so demands care: misuse can potentially damage the board and/or workstation. Since they need not and so should not be used within the context of this worksheet, you should find they are covered by red (for $V_{dd} \equiv 5\text{V}$) and black (for GND) protective jumpers. Leave them in place!

Q1–§3 An overview of the NANDboard: the input group

The input group houses 4 switches (or "push buttons" if you prefer). Each j-th switch is connected to 4 pins (to the right of the switch) labelled X_j: the switch controls the value of those pins, in the sense that

$$X_j = \begin{cases} 0 & \text{if the } j\text{-th switch is not pressed} \\ 1 & \text{if the } j\text{-th switch is pressed} \end{cases}$$

Q1–§4 An overview of the NANDboard: the output group

The output group houses 4 LEDs. Each j-th LED is connected to 4 pins (to the left of the LED) labelled R_j: the LED is controlled by the value of those pins, in the sense that

$$\text{the } j\text{-th LED is } \begin{cases} \text{off} & \text{if } R_j = 0 \\ \text{on} & \text{if } R_j = 1 \end{cases}$$

Q1–§5 An overview of the NANDboard: the NAND group(s)

Each NAND group houses 4 NAND operators[1], meaning there are 16 such operators on the board. Each j-th operator in each i-th group is connected to 4 input pins (to the left of the operator) and 4 output pins (to the right of the operator): the operator uses the input pin values to control the the output pins, in the sense that it computes

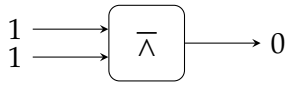
$$r_{i,j} = x_{i,j} \mathbin{\overline{\wedge}} y_{i,j}.$$

In more detail, the behaviour of each operator is as follows:

- Each operator will use 2 inputs to compute an output. As Figure 2 shows, however, said inputs can be provided by *either* of 2 different input pins. For example, notice there are *two* pins labelled $x_{0,0}$ (resp. $y_{0,0}$) which a) are connected together, and b) provide the first (resp. second) of 2 inputs to the operator.
- Although you are free to use either of the 2 input pins labelled $x_{i,j}$ (resp. $y_{i,j}$) to provide a given input, there is one restriction: the pins cannot be contradictory, in the sense that both input pins labelled $x_{i,j}$ (resp. $y_{i,j}$) must have the same value.
- Each output pin reflects the output from an associated operator, and is also visualised by an associated LED[2]: if the output is 0 then the LED is off, whereas if the output is 1 then the LED is on.
- By default, each input pin is 0. This means that when an (empty) board is initially powered-on, each operator will compute $0 \mathbin{\overline{\wedge}} 0 = 1$ and so each LED *should* be on: this fact provides a quick way to check the board is working correctly before carry on!
- To change what a given operator computes, connections can be made between pins using jumper wire. For example, the input pin for an operator can be connected to any (otherwise unused)
  - constant 1 pin (which are above the input pins),
  - input group pin,
  - NAND group output pin, *or*
  - NAND group input[3] pin.
- The NAND groups and hence operators are independent, meaning you can form inter- *and* intra-group connections (i.e., inside one group *and* between two groups).

Q1–§6 Example #1 (red): validating NAND-like behaviour

We already saw that a Boolean expression can be thought of Mathematically, *or* diagrammatically. For example, we can express $1 \mathbin{\overline{\wedge}} 1 = 0$ using the following diagram



instead. Using the NANDboard, we can *concretely* implement and thus evaluate the expression, rather than reason about it *abstractly* via pencil-and-paper alone:

[1] In reality, the operators are realised using **logic gates**; we will encounter the internal, transistor-based design of these components in some later lecture slot(s), but you can ignore this for now.

[2] Keep in mind that the LED colour is *not* relevant: it turns out that the different manufacturing runs that produced our stock of NANDboards have used green, red, or even a mixture of the two!

[3] At first glance this may seem odd (it suggests the input pin is being used as an output), but *can* be useful in some circumstances. Imagine you want to compute $t \mathbin{\overline{\wedge}} t$, for example: one approach would be to connect $x_0 = t$ and $y_0 = t$, whereas another approach would be to connect $x_0 = t$ and $y_0 = x_1 = x_0 = t$. Although the first approach is arguably simply, the second may be advantageous if 1 pin is available to provide a connection from t (e.g., one free output pin).

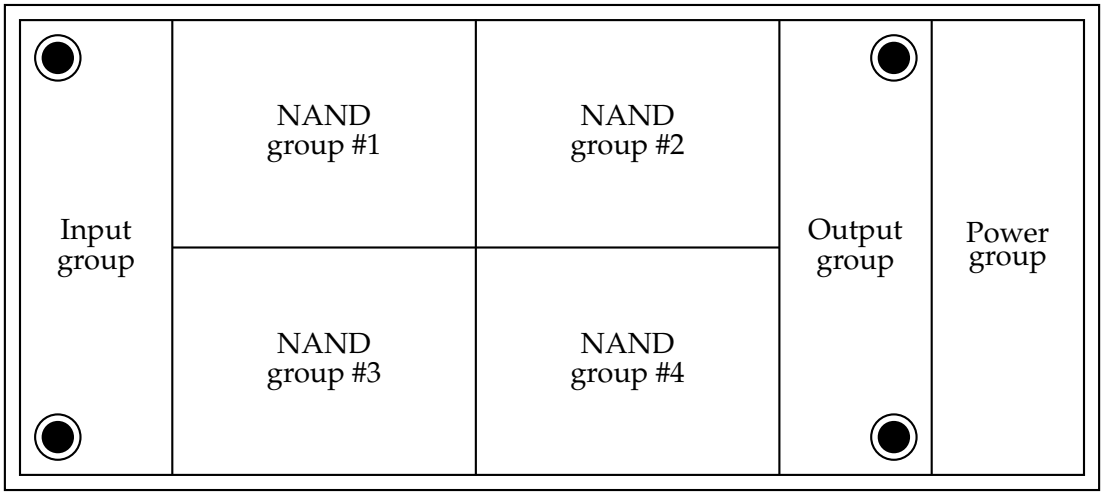


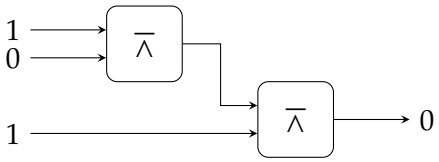
Figure 1: *A high-level overview of the NANDboard.*

a Replicate example #1, using 2 jumper wires the match the (red) connections shown in the top-left NAND group in Figure 2. As soon as you connect both jumper wires, the top-most NAND operator will change from computing $0 \overline{\wedge} 0 = 1$, meaning the LED is on, to $1 \overline{\wedge} 1 = 0$, meaning the LED is off.

b Write down the truth table for NAND, then work step-by-step through each row: verify that the behaviour of operators on the board match the truth table, using jumper wire to connect the input pins to 0 or 1 as appropriate (versus *only* 1 as above).

Q1–§7 Example #2 (blue): using multiple operators

Consider a slightly more complicated example, whereby $(1 \overline{\wedge} 0) \overline{\wedge} 1 = 0$ now includes *two* NAND operators rather than one. The same approach applies to diagrammatic expression, meaning we could draw



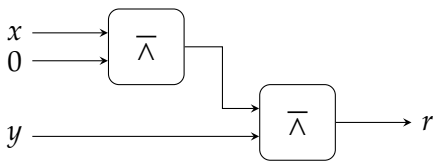
instead.

a Replicate example #2, using 3 jumper wires the match the (blue) connections shown in the top-right NAND group in Figure 2. Notice that the top-most LED is on because $1 \overline{\wedge} 0 = 1$ while the LED below it is off because $(1 \overline{\wedge} 0) \overline{\wedge} 1 = 1 \overline{\wedge} 1 = 0$.

b Verify that you can use either input pin or any output pin for a given operator: alter the design so it evaluates the *same* expression (and so yields the same result) via a *different* configuration of jumper wires.

Q1–§8 Example #3 (green): using switches to model variables

If we write $(x \overline{\wedge} 0) \overline{\wedge} y = r$, the expression has the same *structure* as the previous example, but differs in the sense that some constant values are replaced by a variable. The result obviously depends on x and y (i.e., is a function of the LHS), so the result (or RHS) is also written as a variable r. Again we could draw



instead.

a Replicate example #1, using 3 jumper wires the match the (green) connections shown in the bottom-left NAND group of Figure 2.

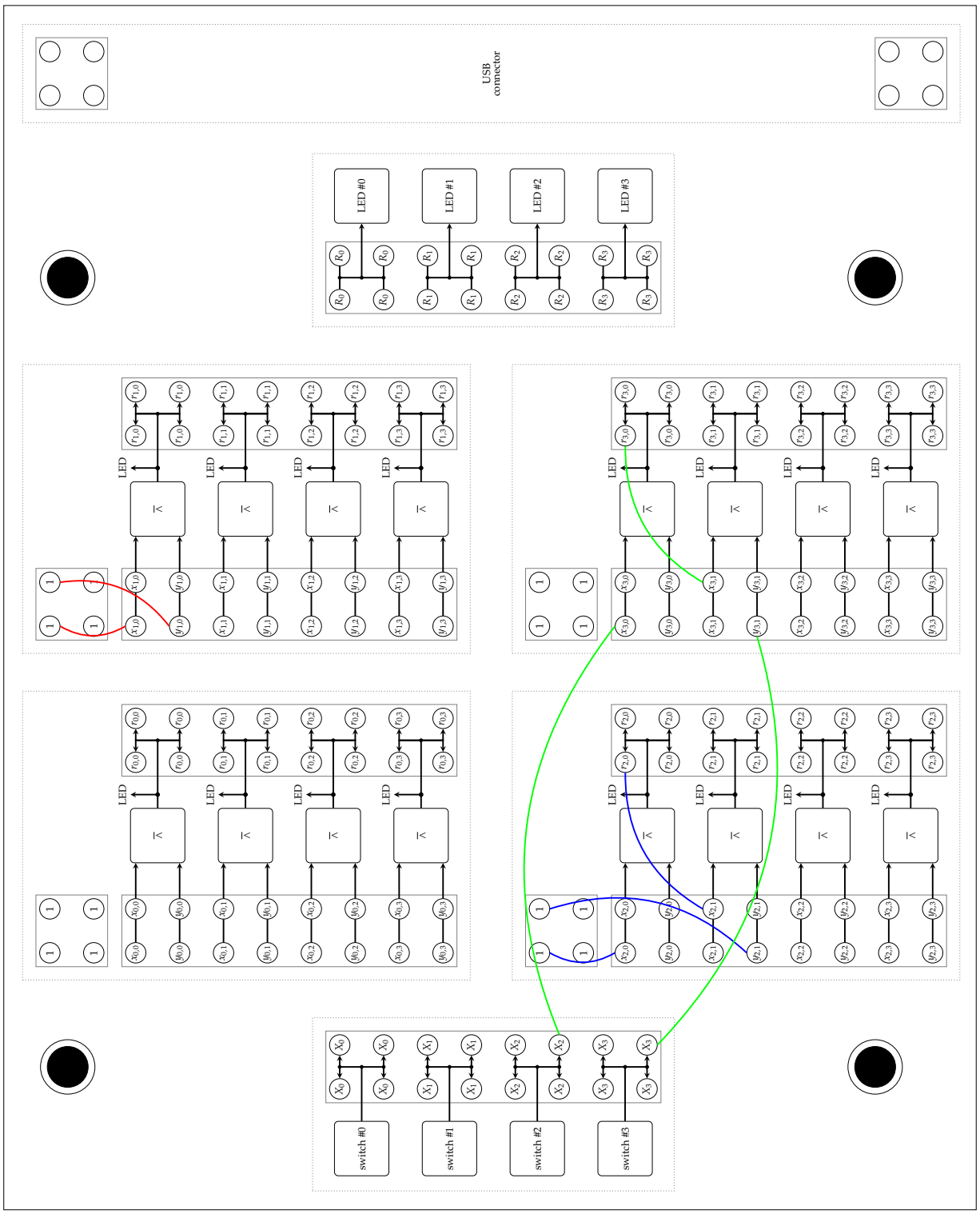


Figure 2: *A low-level overview of the NANDboard: note that this diagram includes three examples, with examples #1, #2, and #3 using the top-right, bottom-left, and bottom-right NAND groups and red, blue, and green wires respectively.*

b Verify that pressing the switches yields the result you expect. For example, if you press *both* switches this would be the analogy of setting $x = y = 1$ and thus producing $r = (1 \overline{\wedge} 0) \overline{\wedge} 1 = 0$ as above.

▷ **Q2[C].** a Use a NANDboard to implement

i NOT,
ii AND,
iii OR, and
iv XOR

operators. As above, verify that the physical computation matches what you expect in theory by work step-by-step through each row in the associated truth table.

b Stated generally, the **majority** function takes n inputs, and produces 1 as an output iff. $\lceil \frac{n}{2} \rceil$ or more of the inputs are equal to 1; otherwise the output is 0.

Consider the specific case of $n = 3$ inputs called a, b, and c, where the function can be written more descriptively as

$$\text{MAJ}(a, b, c) = \begin{cases} 1 & \text{if 3 or 2 of } a, b \text{ and } c \text{ are equal to 1} \\ 0 & \text{if 1 or 0 of } a, b \text{ and } c \text{ are equal to 1} \end{cases}$$

Your goal in this question is to first design then implement this function using a NANDboard. This is more difficult than previous questions, so the recommended approach is to work step-by-step:

- write a truth table for the function to specify the behaviour required,
- think about how this behaviour can be realised using NOT, AND and OR operators, and write this design down on paper,
- translate the design into NAND-based version, i.e., one using only NAND operators, then
- implement the NAND-based version using a NANDboard, and verify that it works correctly.

§2. R-class, or revision questions

▷ **Q3[R].** There is a set of questions available at

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

Using pencil-and-paper, each asks you to solve a problem relating to Boolean algebra. There are too many for the lab. session(s) alone, but, in the longer term, the idea is simple: attempt to answer the questions, applying theory covered in the lecture(s) to do so, as a means of revising and thereby *ensuring* you understand the material.

§3. A-class, or additional questions

▷ **Q4[A].** Imagine you take your NANDboard-based implementation of a majority gate, and connect the output to reuse it as an input. For example,

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

describes a situation where the output is connected to c.

a Experiment with different values of the remaining inputs a and b; write a truth table to capture the behaviour you observe.

b Based on the truth table above, can you think of a use for this component?

▷ **Q5[A].** bOOleO is a card game involving Boolean algebra. Using AND(r), OR(r) and XOR(r) to denote cards for Boolean AND, OR and XOR operators whose output is r, a full deck of bOOleO cards includes the following:

- 48 Boolean operator cards, namely
  - $8 \times$ AND(0) cards,
  - $8 \times$ AND(1) cards,
  - $8 \times$ OR(0) cards,

  - $8 \times$ OR(1) cards,
  - $8 \times$ XOR(0) cards,
  - $8 \times$ XOR(1) cards,
  - $8 \times$ NOT cards

  and

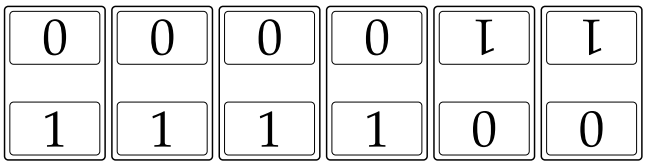
- 6 Boolean value cards (which have a 0 and 1 on).

There is a Wikipedia entry

https://en.wikipedia.org/wiki/Booleo

with a brief overview of the rules, but, for completeness, an alternative follows:

- Decide which player will act as the dealer: they should shuffle the operator cards, and (separately) the value cards. The dealer should then place the value cards between the players (st. the short edges face the players). An example might be as follows:



  The top player faces downward at the value cards (read from left-to right) 110000; the bottom player faces upward at the value cards 111100.

  Finally, the dealer should deal 4 operator cards to each player: this becomes their hand. All remaining operator cards form a draw deck from which players take cards (the draw deck is placed face down, meaning the card types cannot be seen).

- The goal of the game, for each player, is to form a valid tree (or pyramid) of operator cards. Some rules for forming the tree are as follows:

  - For a given player, the tree should start from (i.e., the base should be) the value cards and extend outward towards them.
  - The tree should end with (i.e., the tip should be) a single output matching the right-most value card (in the example, for the bottom player the output should be 0).
  - The inputs and outputs of operator cards in the tree should respect Boolean algebra: if one evaluates the operator (by using inputs from the layer of tree above it), the output should match the following:

x	y	valid operator card		
0	0	AND(0)	OR(0)	XOR(0)
0	1	AND(0)	OR(1)	XOR(1)
1	0	AND(0)	OR(1)	XOR(1)
1	1	AND(1)	OR(1)	XOR(0)

- The game is turn-based, st. one player has a turn then the other and so on until the game terminates. The non-dealer should start. Within their turn, a player

  - draws a new operator card from the draw deck, then
  - plays *or* discards an operator card from their hand

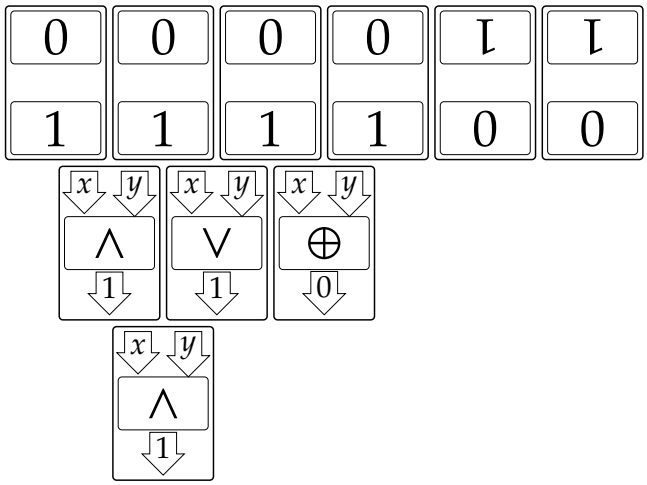
  meaning that after their turn, they *still* have a 4 card hand. If the player plays a card, this implies they will either

  - use a NOT operator card, which flips a value card (literally rotating it) selected by the player,
  - place any other operator card in the tree, thus extending it, or
  - replace any operator card in the tree.

- Some special-cases need further explanation:

  - When a card needs to be discarded, it is placed on a discard deck next to the draw deck. If/when the draw deck is empty, the dealer should shuffle the discard deck which then becomes the new draw deck.
  - When a turn is complete, one or *both* trees need to be reevaluated: any part of either tree which is invalid (e.g., the output of a given operator card no longer matches the inputs), must be discarded. The reevaluation process should start at the value cards, and progress layer-by-layer through the entire tree.
  - An operator card which is left "dangling" with no input(s), e.g., due to an operator card in the layer above having been discarded, is *not* viewed as invalid per se: rather, it is left inactive and then reevaluated when an input becomes available.
  - When a NOT operator card is played, it is removed from play permanently rather than being discarded.
  - When a NOT operator card is played, the opponent can optionally *cancel* the flipping effect by playing a NOT of their own during their turn. The need to wait and see what the opponent does suggests you should wait to see if the tree(s) need reevaluation, and only do so if the value card is flipped after all.

- Ignoring the top player, imagine the game state wrt. the bottom player is as follows:



  Assuming the relevant cards are available, some valid moves and their implication then include:

  - Extend the tree by placing an operator card next to the lower AND(1) card, e.g., XOR(1) or AND(0).
  - Update the tree by replacing the OR(1) operator card, for example: if we replace it with AND(1) then the card below it is still valid so is retained, if we replace it with XOR(0) (which is still valid wrt. the inputs) then the card below becomes invalid and is discarded.
  - Use a NOT operator, on the right-most value card whose value is 1 for example. In this case it will flip from 1 to 0, so the XOR(0) card below becomes invalid and is discarded.

Based on this, have a go at the following tasks:

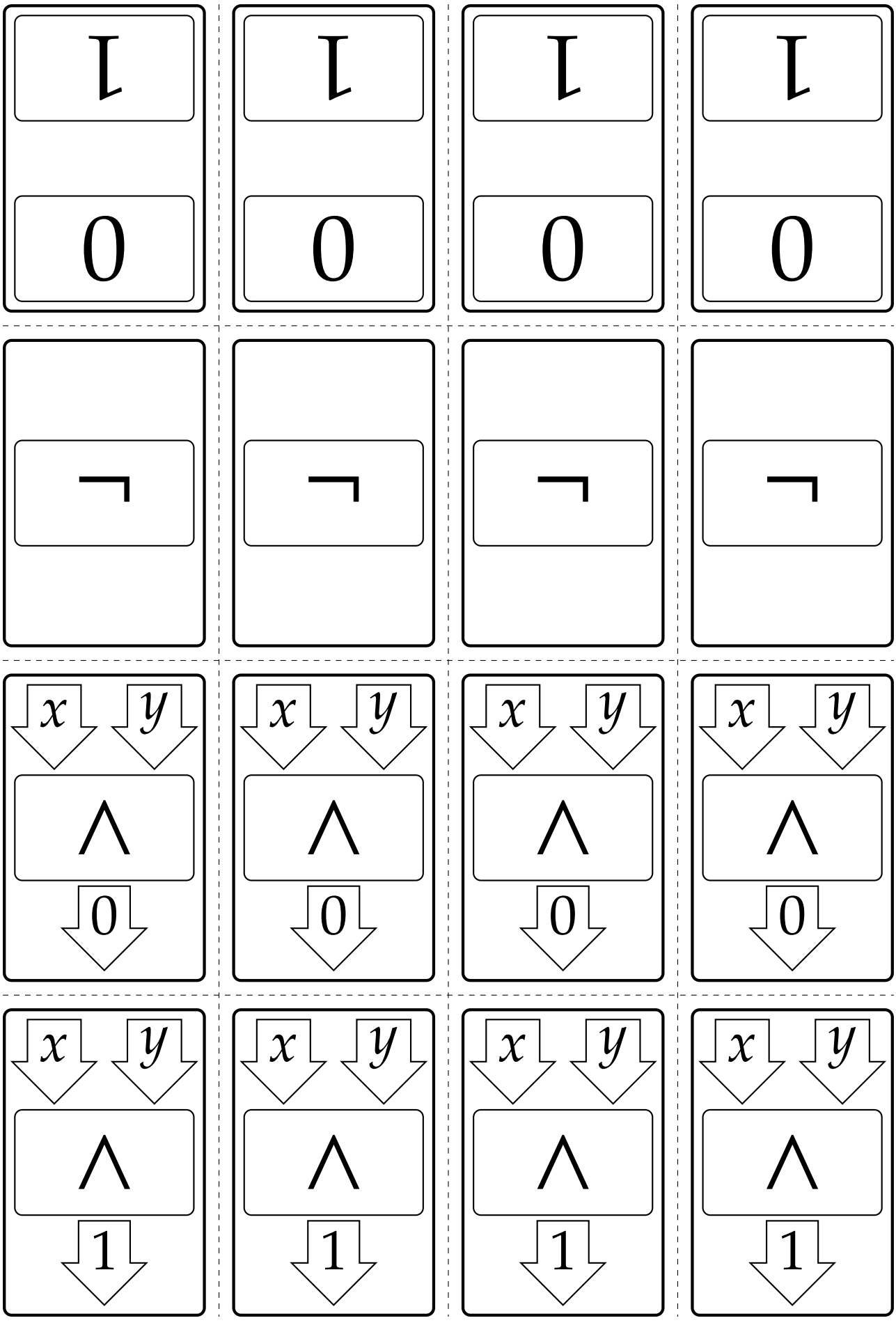
a The pages toward the end of this worksheet include enough cards for a *half* deck: this implies you first need to find an opponent to play against, and then combine your cards. Find another student, and see if you can complete a game of bOOleO. Or, if you prefer, use the Android-based app available at

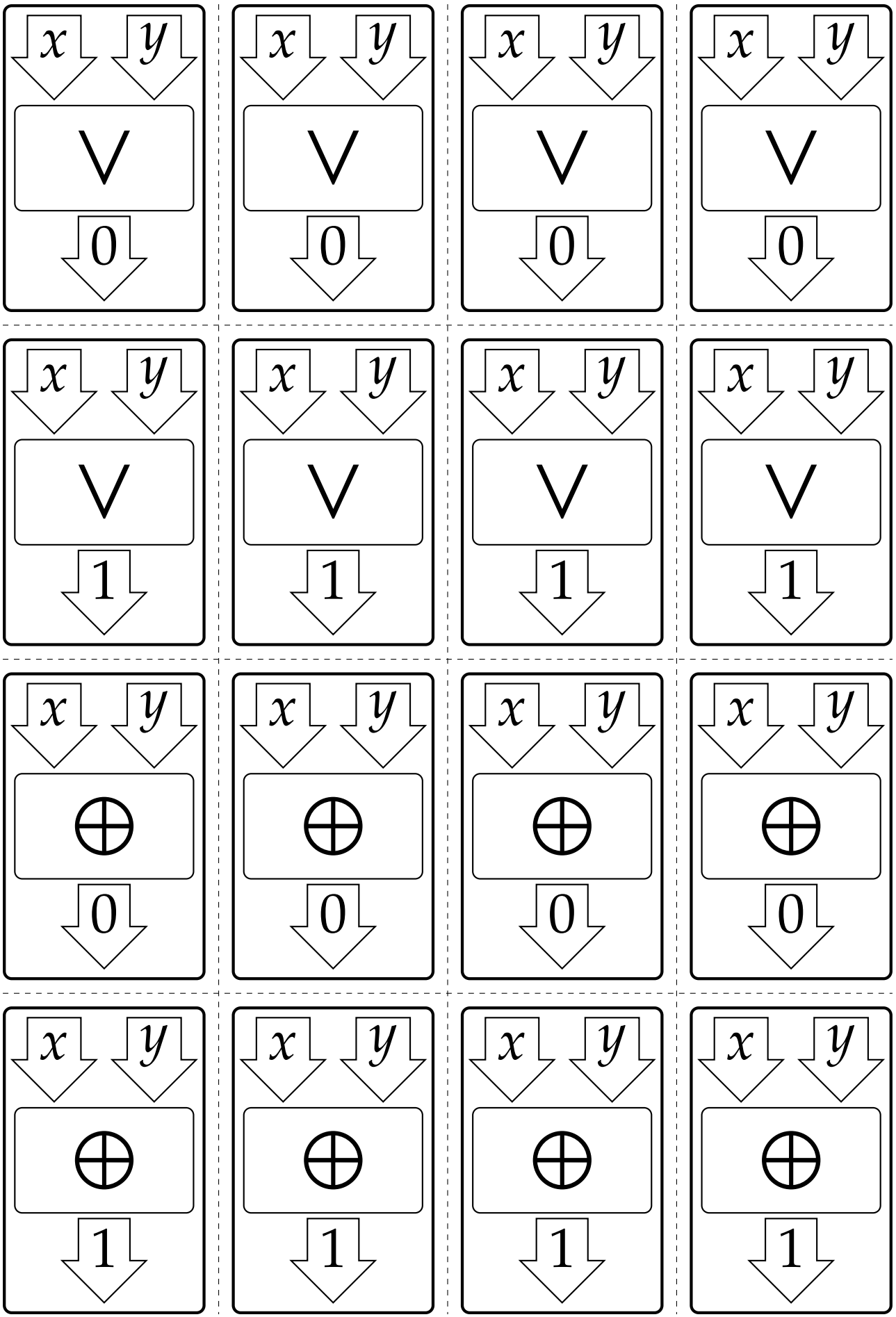
  https://play.google.com/store/apps/details?id=com.oliviercrt.booleo

  instead.

b Based on your experience, think about the following:

  - There are various strategies relating to different aspects of the game. Think about the first row of operator cards placed below the value cards: is there a reason to favour one type over another, and why is this the case?
  - It is possible for one or other player to hit a "dead end" meaning the game cannot terminate (which is quite annoying): can you identify this case, and suggest a way to resolve the problem?
  - Various extensions or variations of bOOleO are possible; bOOleO-N, for example, introduces NAND and NOR operator cards. Can you think of any other interesting variations?
    For example, one *could* imagine a "wildcard" operator of some sort: is there a Boolean or physical justification for such an operator? Or, what happens if you add n-input, m-output operators (for $n > 2$ and/or $m > 1$): can you think of any interesting examples? Or, what about involving more than two players?


⊥
0
¬
x
y
∧
1


x y ∨ 0
x y ∨ 0
x y ∨ 0
x y ∨ 0
x y ∨ 1
x y ∨ 1
x y ∨ 1
x y ∨ 1
x y ⊕ 0
x y ⊕ 0
x y ⊕ 0
x y ⊕ 0
x y ⊕ 1
x y ⊕ 1
x y ⊕ 1
x y ⊕ 1