- 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 #3

§1. C-class, or core questions

▷ **Q1[C].** This and subsequent lab. worksheets make use[1] of Logisim Evolution (referred to using the short-hand LogisimEvo), a tool which supports visual design and simulation of digital logic designs: doing so enables hands-on exploration of associated concepts, and thus aims to enhance your understanding based on the lecture slot(s) alone. Note that although you *could* download and install LogisimEvo yourself, it is not *necessary* to do so because it has been pre-installed, e.g., on the lab. workstations.

A brief introduction. The goal[2] of *this* question is to explain the high-level features in and workflow for using this tool, offering experience that will allow you to engage with tasks and challenges presented in *later* questions. Start by using a terminal[3] to execute LogisimEvo:

a update your `${PATH}`[4] environment variable by executing

```
export PATH="${PATH}:/opt/logisim-evolution/bin"
```

b check said update worked correctly by executing

```
which logisim-evolution
```

noting that any reported error (e.g., `no logisim-evolution in ...` or similar) suggests it did not: ask for help!

c then, finally, execute

```
logisim-evolution
```

You *should* eventually see something similar to Figure 1, which has been annotated to highlight several major features:

- The canvas pane is where we will implement (or "draw") the design itself; doing so is assisted by a grid, which acts to align component instances.
- The tool-bar pane controls the manner or mode in which you interact with the canvas, i.e., what clicking on it does. It includes icons which allow you to
  - alter the state of components (the "hand" or "poke" tool),
  - place, select, and alter the properties of components (the "arrow" or "select" tool),
  - place wires (the "wire" tool),
  - place text-based annotation and documentation (the "text" tool),
  - access common components (e.g., input and output pins, logic gates, etc.) via a short-cut.
- The explorer pane includes 2 tabs:
  - The design tab lists the set of usable components which can be selected, and then placed on the canvas. Toward the top, a list of user-defined components is shown. By default there is one user-defined component called `main`, which you can think of as roughly analogous to the `main` function in a C program: it acts as the top-level (or entry point) in the design. Notice that a magnifying glass annotation shows the current component reflected by the canvas: it can be changed by simply double-clicking on an alternative. Toward the bottom, a tree of built-in components is shown: these are organised hierarchically, so all similar components are listed under a single heading.
  - The simulate tab lists the active simulations. Typically this includes the current design only, but, either way, is only useful for specific tasks or situations somewhat beyond our scope.

[1] See, e.g., https://github.com/logisim-evolution noting that although we focus specifically on LogisimEvo version 3.8.0, use of *other* versions is plausible modulo any minor incompatibilities. Keep in mind, however, that LogisimEvo is *not* compatible with different tools with similar names; a specific, and important example is the now unsupported Logisim tool https://www.cburch.com/logisim from which LogisimEvo was derived (or has evolved, per the name).

[2] The high-level focus implies that a *non*-goal is therefore an in-depth tutorial-style introduction. Rather, there is an emphasis on *you* to, e.g., explore and use the wider documentation as and when need be: see, e.g., the `Help→User Guide` menu item.

[3] That is, within a BASH shell (or prompt, e.g., a terminal window) or similar: see, e.g., https://en.wikipedia.org/wiki/Unix_shell.

[4] http://en.wikipedia.org/wiki/PATH_(variable)

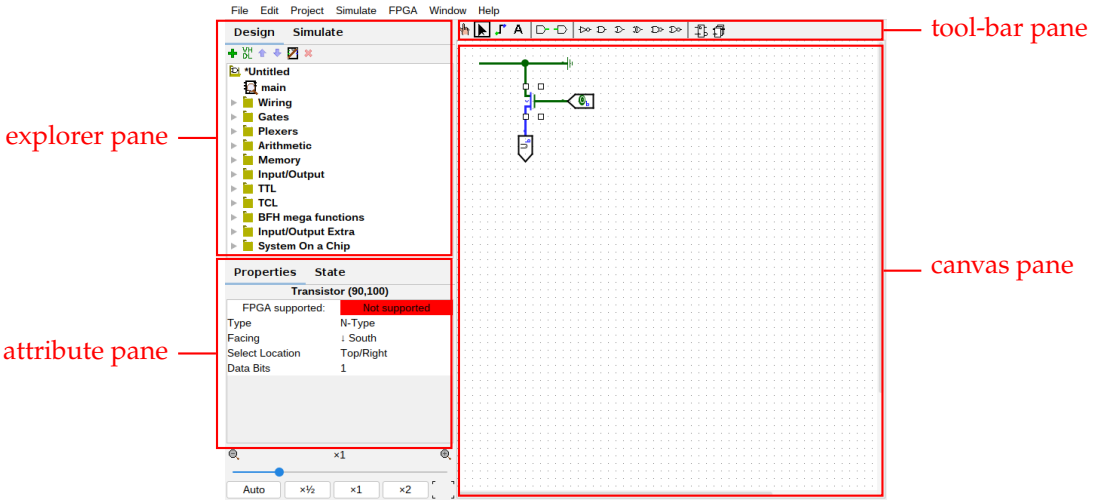
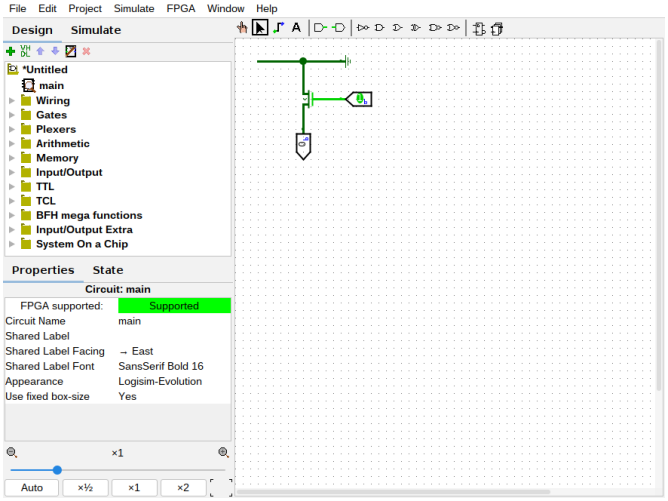
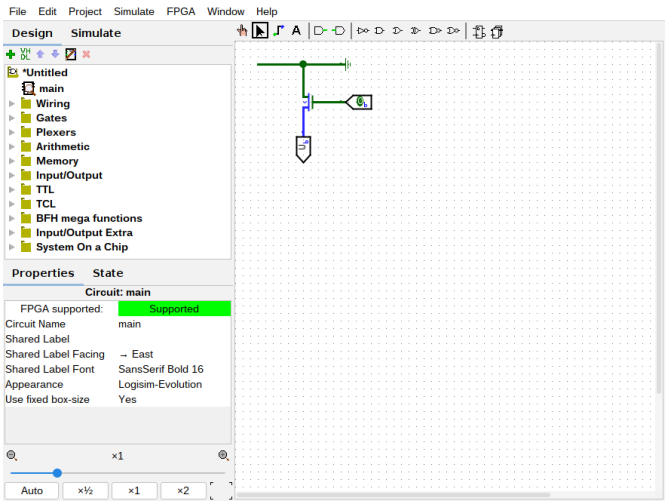


Figure 1: *The LogisimEvo user interface, annotated to illustrate major features.*



(a) *Step #1: input pin value is* 1.



(b) *Step #2: input pin value is* 0.

Figure 2: *An example using LogisimEvo: a V_{ss} (or ground) power rail, an N-MOSFET transistor, an input pin, and an output pin.*

- The attribute pane includes 2 tabs:
  - The properties tab lists a set of properties for the component currently selected on the canvas (if any): each property can be changed, e.g., in order to control the display or behaviour of said component.
  - The state tab lists the values of each register component, and therefore state of active simulation as a whole, iff. the associated property is enabled.

Figure 2a captures a (simple) example, where a single N-MOSFET transistor is placed on the canvas: since the transistor is currently selected (as shown by the square control boxes around the component), fields within the properties tab show it is an N- versus a P-MOSFET and south-facing. This latter attribute is important, because it controls the direction that a connection "through" the transistor can be made (as shown by the an arrow annotation inside the component). Notice that 1) the top terminal is connected to the V_{ss} (or ground) power rail via a wire, 2) the bottom terminal is connected to an output pin, and 3) the gate terminal is connected to an input pin. Keeping the the south-facing direction of this N-MOSFET in mind,

- Figure 2a captures the case where the top and bottom terminals are connected (i.e., V_{ss} and the output pin are connected) as a result of the gate terminal being connected to 1, whereas
- Figure 2b captures the case where the top and bottom terminals are disconnected (i.e., V_{ss} and the output pin are disconnected) as a result of the gate terminal being connected to 0.

Notice that changing the input pin value (using the poke tool) will cause LogisimEvo to (re)simulate the design and thereby update associated intermediate and output values. More specifically, 1) wires on the canvas will change colour to reflect their value, and 2) the output pin will show the value 0 or U (for undefined) so as to reflect connection to or disconnection from V_{ss} respectively. Also notice that the input and output pins can be configured as 3-state (i.e., capable of taking the values 0, 1, or undefined) or not using the properties tab, and can potentially represent larger than 1-bit values if need be (although this is not useful here); in the same way as a transistor, the direction they face in is important because this controls where their connection point is.

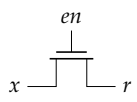
Some initial, exploratory tasks.

a Reproduce and then experiment with the introductory example:

- Place four components on the canvas, namely a V_{ss} (or ground) power rail (i.e., `Wiring→Ground` in the design tab), an N-MOSFET transistor (i.e., `Wiring→Transistor` in the design tab), an input pin (i.e., `Wiring→Pin` in the design tab), and an output pin (i.e., `Wiring→Pin` in the design tab).
- Change the component properties to suit: this means changing, e.g., the output pin type (which defaults to an input rather than an output pin), and the transistor type (which defaults to a P- rather than an N-MOSFET).
- Draw wires to form connections between components.
- Change the input pin value, and observe the associated output.

b Throughout this and subsequent lab. worksheets that use LogisimEvo, any implementation task can be approached by either basing your work on use of 1) your own, user-defined components, and/or 2) analogous built-in components. Once you are confident an implementation functions correctly, it could be viewed as good practice to produce a user-defined component from it; this means you can more easily reuse it, for example. Even so, it is important to stress that *both* approaches are valid. Adopting the latter approach can make sense, because it helps to limit both the volume of work required *and* any dependency between tasks. Adopting the former approach can make sense, because it implies a more complete, end-to-end implementation: if you implement an *entire* design based on transistors (or at least user-defined components based on transistors), for example, there can be no "gap" in your understanding.

Either way, exploring *how* LogisimEvo supports user-defined components is useful because it then enables use of them if/when it makes sense to do so. The example above reflects a so-called enable gate, i.e.,



although currently the input x has been fixed to V_{ss}. Translate your implementation into a more general-purpose (i.e., modelling x with an additional input pin), user-defined and therefore reusable component.

▷ **Q2[C].** In the lecture slot(s), we saw that NOT, NAND, and NOR logic gates can be constructed directly using transistors; more specifically, the transistors are carefully arranged to form 1) a pull-up network of P-MOSFET transistors that can set the output to 1 (by connecting it to the V_{dd} power rail), and 2) a pull-down network of N-MOSFET transistors that can set the output to 0 (by connecting it to the V_{ss} or ground power rail). Apply this theory: use LogisimEvo to implement and simulate a transistor-based

a NOT gate with input x and output r,

b NAND gate with inputs x and y and output r, and

c NOR gate with inputs x and y and output r.

▷ **Q3[C].** In the lecture slot(s), we saw that NAND and NOR are universal, in the sense that they can be used to construct a set of standard Boolean operators and hence logic gates, i.e., NOT, AND, and OR. Apply this theory: noting one *could* focus on NOR instead, use LogisimEvo to implement and simulate a NAND-based

a NOT gate with input x and output r,

b AND gate with inputs x and y and output r,

c OR gate with inputs x and y and output r, and

d XOR gate with inputs x and y and output r.

▷ **Q4[C].** In the lecture slot(s), we covered the design of a ripple-carry adder which can compute the sum $r = x + y$ for n-bit summands x and y; it does this by mirroring the process of long-hand addition, using n full-adder instances linked by a carry chain. Use LogisimEvo to implement and simulate such a component: assuming $n = 4$, work step-by-step by focusing on

a a 1-bit full-adder component, then

b a 4-bit ripple-carry adder component.

§2. R-class, or revision questions

▷ **Q5[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.

▷ **Q6[R].** There is a huge, diverse set of online resources relating to digital logic in general, and NAND- and NOR-based design and implementation specifically. An example is

https://www.nandgame.com

which arguably offers an attractive alternative to use of Logisim, aligned with the content and ethos of Nisan and Schocken [1]: using the tool to work through the associated exercises is an great way to practice and revise this topic.

§3. A-class, or additional questions

▷ **Q7[A].** Throughout the lecture slot(s), we focused on implementing logic gates using transistors. The motivation for doing so is the current dominance of this approach in micro-electronics: it would be somewhat unusual for you to own a device *not* based on some form of transistor.

However, it is important to see that Boolean algebra (and hence computation) is *independent* from this implementation technology. Put another way, we *could* realise the Boolean operators using a technology *other* than transistor-based logic gates; the result would be a computer, although perhaps not with the same characteristics we are used to. Examples of "other" technologies include

- pulleys, e.g.,
  https://www.youtube.com/watch?v=CNbScb8v-MI
- fluids, e.g.,
  https://www.youtube.com/watch?v=NKxFG21vw_I
- lights, e.g.,
  https://www.youtube.com/watch?v=eFhgb5CqAy8
- dominoes, e.g.,
  https://www.youtube.com/watch?v=OpLU__bhu2w

Based on this,

a try to assess how (and why) the characteristics of a computer based on these examples might differ from one that uses transistors (e.g., list any obvious advantages or disadvantages of each), then

b try to think of (or find) any other viable "other" technologies.

▷ **Q8[A].** In the previous questions, you implemented 2-input, 1-output NOR, NAND, OR, and AND gates. It is common, however, to require gates with *more* than 2 inputs: we might want to compute the NAND of x, y, and z for example.

a Write out a truth table that specifies the behaviour you expect for a 3-input NAND gate.

b A natural first attempt might use two 2-input NAND gates to compute

$$x \overline{\wedge} y \overline{\wedge} z,$$

but this is incorrect: prove this is the case, and explain why (versus a similar scenario using AND instead of NAND, for example).

c Irrespective of the above, first design, then use LogisimEvo to implement and simulate a *correct* 3-input NAND gate directly using MOSFET transistors.

d If you needed an n-input, 1-output NAND gate, for example, can you articulate the general design strategy required (i.e., the pattern that emerges in terms of how the transistors are arranged)?

▷ **Q9[A].** In the lecture slot(s) we discussed *two* methods for producing a Boolean expression (in SoP form) from a truth table: before the more human-friendly Karnaugh map, we looked at a more machine-friendly algorithm. This question tasks you withimplementing said algorithm within a function whose prototype is

```
void table2expr( int n, char* v[], bool* t );
```

The arguments should allow the caller to specify

- `n`, the number of variables,
- `v`, the names of variables, i.e., `v[ i ]` for $0 \leq$ `i` $<$ `n` is the `i`-th variable name, and
- `t`, the truth table outputs, i.e., `t[ i ]` for $0 \leq$ `i` $< 2^{\texttt{n}}$ is the function output in the `i`-th truth table row

and the output (i.e., the expression) should be printed to `stdout`.

References

[1] N. Nisan and S. Schocken. *The Elements of Computing Systems: Building a Modern Computer from First Principles*. MIT Press, 2005. URL: http://www.nand2tetris.org (see p. 5).

A Frequently Asked Questions (FAQs)

The transistor I'm using in my design don't work as expected: why not?! There are clearly other possibilities, but two (very) common problems are outlined below; fortunately, once you identify such a problem the solution is simple in both cases.

1. A given MOSFET transistor has a facing property, which controls the direction that a connection "through" the transistor can be made: this direction is shown by an arrow annotation on the component itself. You could think about this as a way to tell LogisimEvo which terminal is the source, and which is the drain. This is useful because it means we can swap them freely, which renders the component easier to use (even though the terminals are pre-defined for standard symbols). This is important, because if a transistor is facing in the wrong direction then the connection and hence output will not work as expected.
2. LogisimEvo models MOSFET transistors with correct *digital* semantics: this represents an abstraction of the underlying implementation to some extent, whereby 1) an N-type MOSFET can drive the output to V_{ss} (or ground, i.e., 0) but not to V_{dd} (or power, i.e., 1), whereas 2) a P-type MOSFET can drive the output to V_{dd} (or power, i.e., 1) but not to V_{ss} (or ground, i.e., 0). This is important, because if you use the wrong transistor type in a given context then the connection and hence output will not work as expected.

I've notice that different wires have different colours: what is the meaning of this? The LogisimEvo documentation has a comprehensive answer to this question: under the the `Help→User Guide` menu item, see the `Guide to Being a Logisim User→Additional features→Wire colors` entry. In short, a light green wire means 1, a dark green wire means 0, a blue wire means X or disconnected (which basically matches the concept of a high impedance value, i.e., what we denote using $\mathbf{Z}$), a red wire means E or error (which basically means the value cannot be determined, e.g., because the wire is driven simultaneously by two *different* values) an orange wire means a data type mismatch (e.g., a n-bit wire is connected to an input expected m bits where $n \neq m$, which is analogous to a type error in C), a black wire means this is a vector of wires (i.e., an n-bit rather than 1-bit value).

I'm struggling to understand and use the concept of user-defined components: help! The LogisimEvo documentation has a comprehensive answer to this question: under the the `Help→User Guide` menu item, see the `Guide to Being a Logisim User→Subcircuits` entry. In short,

1. use the `Project→Add Circuit` menu item to create a new user-defined component, and give it a meaningful identifier,
2. double-click on the user-defined component, which is now listed towards the top of the design tab: this updates the canvas to reflect the new (but currently empty) user-defined component design,
3. implement the user-defined component design (noting it is possible to copy-and-paste it from elsewhere, e.g., the canvas associated with another component), making sure to give the input and output pins meaningful identifiers,
4. use the `Project→Edit Circuit Appearance` menu item to edit how the user-defined component is rendered (e.g., the border size and shape, where the input and output pins are located, etc.),
5. double-click on the main component, which is listed towards the top of the design tab: use the user-defined component like any other, using a single-click to select and place an instance of it on the canvas.

I created a user-define enable gate as suggested, but it doesn't work as expected: help! There are clearly other possibilities, but one (very) common problem relates to the concept of 3-state logic. By default, an input and output pin is *not* 2-state: they take the value 0 or 1 only. When the enable gate is *disabled*, however, i.e., when $en = 0$, the N-type MOSFET will prevent a connection between x and r. As a result, the pin modelling r must have the 3-state property set to **true**; otherwise it will be unable to reflect a disconnected or high-impedance value.