COMS12200 lab. worksheet: week #6

- We intend this worksheet to be attempted in the associated lab. session, which represents a central form of help and feedback for this unit.
- The worksheet is not *directly* assessed. Rather, it simply provides guided, practical exploration of the material covered in the lecture(s): the only requirement is that you archive your work (complete or otherwise) in a portfolio via the appropriate component at

  `https://wwwa.fen.bris.ac.uk/COMS12200/`

  This forms the basis for assessment during a viva at the end of each teaching block, by acting as evidence that you have engaged with and understand the material.
- The deadline for submission to your portfolio is the end of the associated teaching block (i.e., in time for the viva): there is no requirement to complete the worksheet in the lab. itself (some questions require too much work to do so), but there is an emphasis on *you* to catch up with anything left incomplete.
- To accommodate the number of students registered on the unit, the single 3 hour lab. session shown in your timetable is split into two $1\frac{1}{2}$ hour halves. You should only attend *one* half, selecting as follows:
  1. if you have a timetable clash that means you *must* attend one half or the other then do so, otherwise
  2. execute the following BASH command pipeline

  ```
  id -n -u | sha1sum | cut -c-40 | tr 'a-f' 'A-F' | dc -e '16i ? 2 % p'
  ```

     e.g., log into a lab. workstation and copy-and-paste it into a terminal window, then check the output: 0 means attend the first half, whereas 1 means attend the second half.
- Keep in mind that hardware and software within the lab. is managed by the IT Services Zone E team. If you encounter a problem (e.g., a workstation that cannot be powered-on, an error when executing some software, some missing software, or you just cannot log into your account), *they* can help: either talk to them in room MVB-3.41, and/or submit a service request online via

  `http://servicedesk.bristol.ac.uk`

The Logisim project

`http://www.cs.bris.ac.uk/home/page/teaching/material/arch_new/sheet/lab-6.q.circ`

provides a 2-phase clock[1] component: the outputs `phi_1` and `phi_2` represent Φ_1 and Φ_2, i.e., the two non-overlapping clock signals. Download and use[2] this project as a starting point, adding to it within each of the following questions.

Q1. In the lecture(s) we saw how an initial SR-type latch design could be refined, step-by-step, to produce a D-type latch; the latter is attractive, in that it includes an enable signal and avoids problems of meta-stability. However, the design covered was NOR-based:

- research and then formulate an alternate, NAND-based design,
- use the built-in logic gates provided by Logisim to produce a simulated[3] implementation, then
- package your implementation as a Logisim sub-component to allow easy reuse, and
- translate the design into NAND-only form, and produce a physical implementation using your NAND board

[1] Although there are various ways to realise such a component, this design is based on [1, Figure 5.73].

[2] The `Simulate` menu allows management of any clocks present within a given project, including instances of the 2-phase clock provided (which houses a built-in Logisim clock component internally). The `Tick Once` menu item manually provokes a clock transition, from 0 to 1 or 1 to 0, moving simulation ahead in time by one half-cycle. In contrast, the `Ticks Enabled` item enables/disables automatic clock transitions (analogous to automatically selection of `Tick Once` again and again forever); once enabled, the `Ticks Frequency` can be used to control the clock frequency. Although the latter could be deemed more realistic, the former is useful while debugging a design: it allows easier inspection of intermediate state, and hence controlled verification that the design does what it should at each step.

[3] The task of debugging designs that contain sequential components, and whose state changes (semi-)automatically as the result of the features in a clock, can be tough. As with debugging software, a good approach to understanding why some output is incorrect is to first ensure you have a clear picture of how it was computed. The Logisim probe component can be really useful: it allows the inspection of a wire value without adding a "dummy" output. The component will adapt to whatever wire type it is connected to, and allows output in a range of representations (e.g., binary, signed or unsigned decimal, or hexadecimal).

Q2. The motivating example for sequential logic presented in the lecture(s) was that of a cyclic n-bit counter: the goal was to have the output r step through values $0, 1, \ldots, 2^n - 1, 0, 1, \ldots$. Using the 2-phase clock component provided, plus

- the full-adder component developed for the worksheet in week #5, and
- the D-type latch component developed in the previous question,

or equivalent built-in[4] components, implement a simulated 4-bit counter using Logisim.

Q3. This question is a (or perhaps *the*) textbook example of FSM design: it challenges you to design then implement and simulate a traffic light controller in Logisim. The scenario focuses on two roads (a main road and some sort of access road) that intersect; the intersection is managed by two sets of standard, UK-style "red, amber and green" traffic lights. The traffic light controller should

- stop cars crashing into each other, so the behaviour should see
  - green on main road and red on access road, then
  - amber on main road and red on access road, then
  - red on main road and amber on access road, then
  - red on main road and green on access road, then
  - red on main road and amber on access road, then
  - amber on main road and red on access road,

  and then cycle, and
- allow an emergency stop button to force red on both main and access roads while pushed, then reset the system into an initial start state when released.

This is harder than the previous questions, so the recommended approach is to work step-by-step:

- enumerate the inputs and outputs from the controller, and the states it can be in,
- design a FSM (e.g., diagrammatically) that describes all valid transitions between states,
- translate the transition and output functions δ and ω into sets of Boolean expressions, then finally
- fill in the generic framework outlined in the lecture(s) and hence implement the design in Logisim.

Q4[+]. This is an extended rather than core question: it caters for differing backgrounds and abilities by supporting study of more advanced topics, but is therefore significantly more difficult and open-ended. As such, you should *only* attempt the associated tasks having completed all core questions (which satisfy the unit ILOs) first; even then, there is no shame in ignoring the question, or deferring work on it until later. Note that a solution will not *necessarily* be provided.

The previous question presents a somewhat basic scenario. Extend your implementation to cater for one of the following (or another extension):

a If temporary traffic lights are erected to support maintenance work, their timing is crucial wrt. traffic flow. Some roadworks in one lane often dictate use of traffic lights to control shared access to the other lane; their timing (e.g., the period a light is green) should clearly relate to the length of roadworks (e.g., a user-controlled value n).

b Some advanced traffic light controllers support somewhat dynamic, rather than purely statically defined, behaviour. Imagine a car were to approach the main road with no car waiting on the access road, for example; to prevent the car waiting, the controller may detect such a situation (e.g., via a camera) and skip straight to green on the main road and red on the access road.

References

[1] N.H.E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective*. Addison-Wesley, 2nd edition, 1993.

[4] `Arithmetic`→`Adder` and `Memory`→`S-R Flip-flop`, which can be found via the explorer pane, are suitable replacements for your implementations of the full-adder and D-type latch. For the former, you can instantiate an entire 4-bit ripple-carry adder *or* four separate 1-bit full-adder instances; in either case, take care to set the number of bits correctly in the attributes pane. For the latter, it is *vital* you change the component into a latch (rather than flip-flop) by setting the trigger to "high level" rather than "rising edge".