• 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 #11

During the period of time aligned with this lab. worksheet, there is an active (or open) coursework assignment for the unit. You could address this fact by dividing your time between them. However, our (strong) suggestion is to view the former as of secondary importance (or optional, basically), and instead focus on the latter: since it is credit bearing, the coursework assignment should be viewed as of primary importance. Put another way, focus exclusively on completing the latter before you invest any time at all in the former.

§1. C-class, or core questions

▷ **Q1[C].** In the lecture slot(s), we studied a simple, Princeton (aka. von Neumann) style computer based loosely on the Electronic Discrete Variable Automatic Compute (EDVAC); the idea was to introduce the concept of a stored program architecture. Recall that the design included an accumulator called A, a program counter called PC, a memory called MEM used to store data *and* instructions, and an instruction set as follows:

- 00$nnnn$ means nop.
- 10$nnnn$ means halt.
- 20$nnnn$ means A $\leftarrow n$.
- 21$nnnn$ means MEM$[n] \leftarrow$ A.
- 22$nnnn$ means A $\leftarrow$ MEM$[n]$.
- 30$nnnn$ means A $\leftarrow$ A + MEM$[n]$.
- 31$nnnn$ means A $\leftarrow$ A − MEM$[n]$.
- 32$nnnn$ means A $\leftarrow$ A $\oplus$ MEM$[n]$.
- 40$nnnn$ means PC $\leftarrow n$.
- 41$nnnn$ means PC $\leftarrow n$ iff. A = 0.
- 42$nnnn$ means PC $\leftarrow n$ iff. A $\neq$ 0.

a Making appropriate assumptions as required, write a C program which emulates the behaviour (i.e., performs the fetch-decode-execute cycle) of this design: use example programs from the lecture slot(s) in order to verify your implementation functions correctly.

b Using your emulator, write a program (i.e., specify the initial memory content) to compute the sum of n elements held in an array X. That is, compute

$$r = \sum_{i=0}^{n-1} X_i$$

for some known n where X is located at some known address, say m.

§2. R-class, or revision questions

▷ **Q2[R].** There is a set of questions available at

https://assets.phoo.org/COMS10015_2025_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

▷ **Q3[A].** Later in the unit, we study the topic of assembly language[1] and assemblers. In essence, (an) assembly language is a low-level programming language that allows the programmer to express individual instructions; an assembler is a tool that processes an assembly language input program, and, e.g., produces machine code as output. Use of assembly language, and so an assembler, can be attractive, because it 1) can permit greater control than, e.g., a high-level programming language (because the translation from assembly language to machine code is more direct), and 2) reduce the effort required to manually write machine code programs.

a Design an assembly language for the instruction set presented in Question 1. That is, invent a human-oriented syntax (or format) which allows the programmer to express each instruction.

[1] https://en.wikipedia.org/wiki/Assembly_language

b Implement an assembler for your assembly language: reading from `stdin` and writing to `stdout` for example, it should accept a assembly language program as input and produce a machine code program as output. Doing so implies translating, or encoding each human-oriented instruction description (assuming, e.g., one instruction per line) in the former into the corresponding machine-oriented instruction description (i.e., machine code) in the latter.

Although each challenge above is open-ended, consider an example that acts as a starting point:

$$\underbrace{\overbrace{\texttt{A <- A + MEM[10]}}^{\text{human-oriented}} \equiv \overbrace{300010}^{\text{machine-oriented}}}_{\substack{\text{syntax}\\ \equiv \\ \text{form}}} \mapsto \underbrace{\mathsf{A} \leftarrow \mathsf{A} + \mathsf{MEM}[10]}_{\substack{\text{semantics}\\ \equiv \\ \text{function}}}$$

The idea is that the left-hand side is an assembly language description of some instruction, whereas the right-hand side is a machine code description of that instruction the programmer writes the former as part of an assembly language program, and uses an assembler to translate it into the latter.

▷ **Q4[A].** The term instruction set simulator[2] is often used as a more formal description of the emulator you implemented as part of Question 1. The by-design focus of such a tool represents a trade-off, so that, e.g., it 1) allows easy experimentation with an instruction set, and programs written against it, but 2) focuses on functional correctness and so abstracts various details of a concrete implementation (therefore contrasting with a micro-architecture simulator[3] which, e.g., models cycle accurate execution latency).

a Consider the fact that, typically, the number of instructions executed is correlated with execution latency and therefore efficiency (with respect to time): focused on the goal of reducing the number of instructions executed to realise some functionality, design an extension for the instruction set presented in Question 1, i.e., define and implement support for some addition instructions you think will be useful. One potential starting point is the concept of an addressing mode[4], of which the existing instruction set supports one, simple instance.

b How can you evaluate your instruction set extension. What metrics are relevant, for example, and how does your design compare (with respect to those metrics) with any viable alternatives?

[2] https://en.wikipedia.org/wiki/Instruction_set_simulator
[3] https://en.wikipedia.org/wiki/Microarchitecture_simulation
[4] https://en.wikipedia.org/wiki/Addressing_mode