ASSESSED **INDIVIDUAL WORK**

COMP1007
LAB EXERCISE #1

Steven R. Bagley

Submission Deadline:
02/11/2021

Version: 1.00

In this lab exercise, you will implement five logic gates using the **nand2tetris** hardware simulator and hardware description language. Test scripts are supplied for all gates to enable you test whether your implementation is correct. It is strongly encouraged that you develop the gates in the order specified.

This document outlines how the exercise is assessed, and then outlines of the logic gates your are to implement in detail. The skeleton HDL files and test scripts can be found in the following git repository:

`https://projects.cs.nott.ac.uk/2021-COMP1007/2021-COMP1007-LabExercise01`

Good luck…

You will almost certainly find it impossible to complete these exercises unless you have read the relevant chapters of the book '*The Elements of Computing Systems: Building a Modern Computer From First Principles*' — the book which accompanies **nand2tetris**. You can either purchase the book via Amazon or find the relevant chapters on their website at:
`https://www.nand2tetris.org/course`

Implementation Detail

Inside the git repository, you forked and cloned via `git` in the usual fashion, you will find skeleton `.hdl` files and test scripts for each of the components you need to implement. It is **strongly recommended** that you use these skeleton otherwise your circuits may not work against the marking test scripts.

Assessment Notes

The pipeline will mark this coursework automatically by running the test scripts against your implementation of the logic gates. Marks for each gate are shown alongside the description for the gates and will be assigned on a pass/fail basis — if your gate passes the test script, you will be awarded the marks, if it doesn't, you won't. The marks will then be combined to give a final mark out of 10.

Combined, the lab exercises for COMP1007 form part of your programming portfolio will account for 10% of the mark for the module.

Gate Descriptions

There are five logic gates that you need to implement for this lab exercises and they centre around logic circuits that can perform arithmetic on binary numbers. You may well find it instructive to have watched the videos on Moodle for week three. The implementation of the gates is also broadly similar to the gates that form part of **nand2tetris** project two, so you might also want to look there for inspiration.

The table below lists the five gates that you need to create and their function. For both the HalfSubtractor and FullSubtractor, you can find additional information on the wikipedia entry:

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

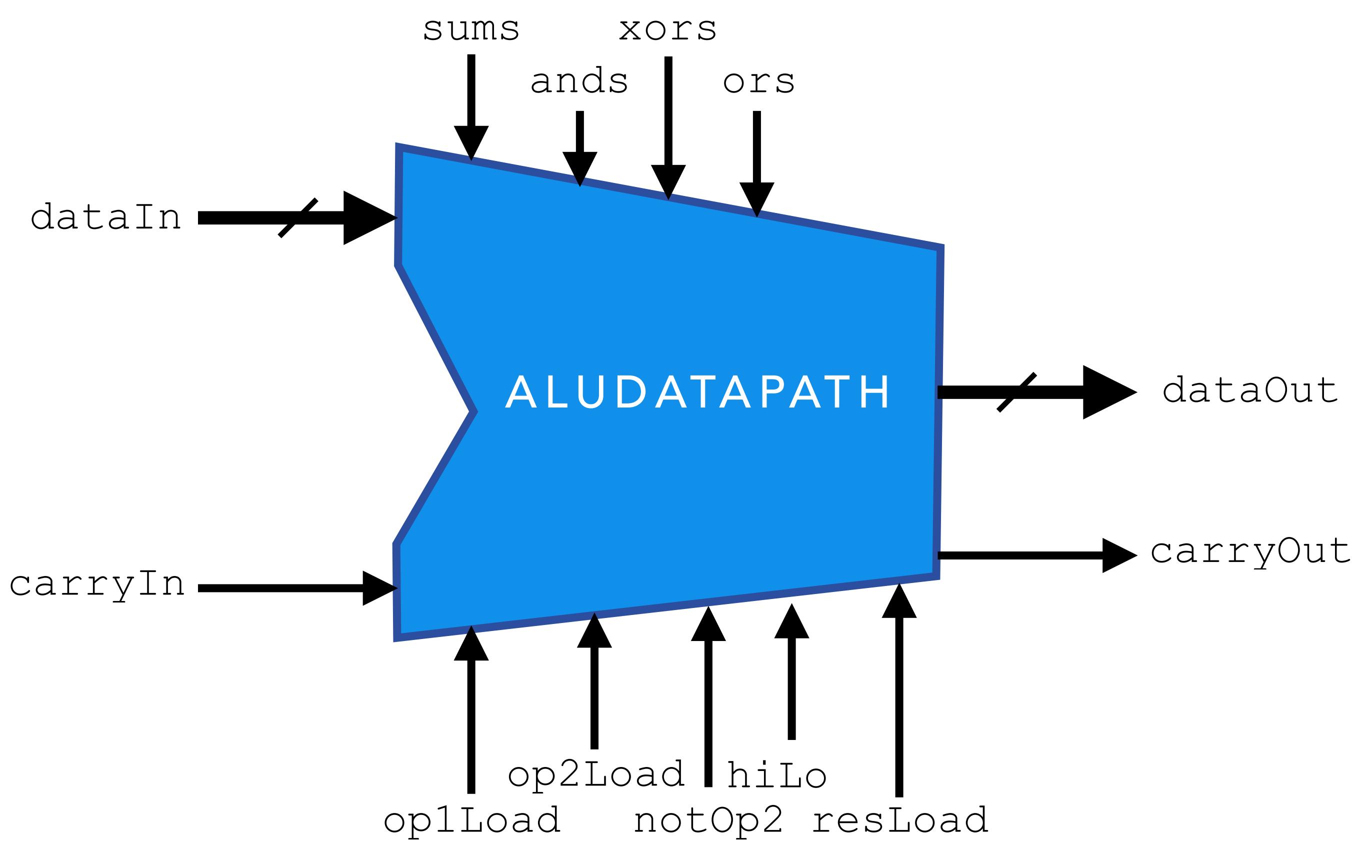
Gate	Description
`HalfSubtractor`	This gate has two inputs, `x` and `y`, and should calculate the result of subtracting `y` from `x`. It should produce two outputs, `d`, the result of the subtraction (also known as the *difference*), and `bout`, the borrow out signal (which indicates whether this calculation needs to borrow one). A truth table for this gate can be found in the slides accompanying the videos for week three.
`FullSubtractor`	This gate is similar to the `HalfSubtractor`, but has an additional input `bin`, which specifies when the previous digit borrowed one from the input `x`. Therefore, this gate is effectively calculating `x - y - bin`. Again, there are two outputs, `d` and `bout` which have the same meaning as in the previous gate. A truth table for this gate can again be found in the slides accompanying the videos for week three.
`Add4`	This has two input buses, `a` and `b` and one output bus, `out`. Each bit of the output should be the result of *adding* together the corresponding input bits in `a` and `b`, while making sure that any carry is propagated to the next bit. Effectively, you are implementing a 4-bit ripple-carry adder. **Note:** You can assume that the single bit `FullAdder` is defined on the system already and takes three inputs, `a`, `b` and `c` (where `c` is the carry *input*), and produces two outputs: `sum.`, which is the result of adding the two bits together and `carry`, which is any carry *produced* by the addition.
`Add4C`	This is an extended version of the Add4 gate above that also has an additional `carryIn` input, which should be used to feed the carry into first addition, and a `carryOut` output which carries the carry out of the final addition of your ripple carry adder.

Gate	Description
`Sub4`	This has two input buses, `a` and `b` and one output bus, `out`. Each bit of the output should be the result of *subtracting* the corresponding input bit in `b` from the corresponding bit in `a`, while making sure that any borrow is propagated to the next bit. The chip also has an output `bout`, which carries the borrow out (bout) of the final subtractor. **Note:** You should use your implementation of a `FullSubtractor` to implement this gate, and you should find the structure is similar to that of `Add4`.

Note As well as connecting the inputs and outputs of a gate to other gates (or the input and output of the `CHIP` you are designing, you can also specify that an input has a specific value by using the keywords `false` and `true`, outputs can also be omitted if you do not wish to connect them to anything)

Predefined gates

Several logic gates are provided for you as part of the **nand2tetris** package (for things like, `And`, `Or` and `Not` etc.) and it is expected that you will make use of them. All gates described in the **nand2tetris** book have pre-defined versions available.


sums
xors
ands
ors
dataIn
ALUDATAPATH
dataOut
carryOut
carryIn
op2Load
hiLo
op1Load
notOp2
resLoad

`Not4`	This has one input bus, `in`, and one output bus, `out`. Each bit of the output is the inverse (i.e. not) of the corresponding input bit.
`And4`	This has two input buses, `a` and `b`, and one output bus, `out`. Each bit of the output is the result of logically *anding* together the corresponding input bits in `a` and `b`.
`Or4`	This has two input buses, `a` and `b`, and one output bus, `out`. Each bit of the output is the result of logically *oring* together the corresponding input bits in `a` and `b`.
`Xor4`	This has two input buses, `a` and `b`, and one output bus, `out`. Each bit of the output is the result of logically *Xoring* together the corresponding input bits in `a` and `b`.
`Add4`	This has two input buses, `a` and `b`, and one output bus, `out`. Each bit of the output is the result of *adding* together the corresponding input bits in `a` and `b`, while making sure that any carry is propagated to the next bit. `Add4` also has an additional `carryIn` input, which is used to feed carry into first addition, and a `carryOut` output which carries the carry out of the final addition.