ASSESSED INDIVIDUAL WORK

COMP1007
LAB EXERCISE #5

Steven R. Bagley

Submission Deadline:
03/12/2021 15:00

Version: 1.00

In lab exercise three, you created a series of gates that could process 8-bit values (`Not8`, `And8`, `Or8`, `Mux8`, etc.) while in lab exercise one you created a single gate, (`Add4C`) which added two 4-bit numbers together. In this exercise, you will combine these gates together to form the Arithmetic and Logic Unit (ALU) for an 8-bit CPU. This ALU is based on the ALU in the 6502 microproccessor[1], and is much simpler to understand than the **nand2tetris** ALU we considered in the online session on the 27th October. Test scripts are supplied to enable you test whether your implementation is correct, don't worry if you didn't complete lab exercise one or three — the git repository contains implementations of all the gate you need to use for this exercise.

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-LabExercise05`

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 implementation may not work against the marking test scripts.

Assessment Notes

The pipeline for this exercise will open on Tuesday 30th November, until then please use the `ALUcore.tst` script to test your implementation in the hardware simulator.

The pipeline will mark this coursework automatically by running various test scripts against your implementation of the `ALUcore` logic gate. Marks for each test are shown alongside the description of the test and will be assigned on a pass/fail basis — if your gate passes the test, you will be awarded the marks, if it doesn't, you won't. The pipeline this time will also be awarding

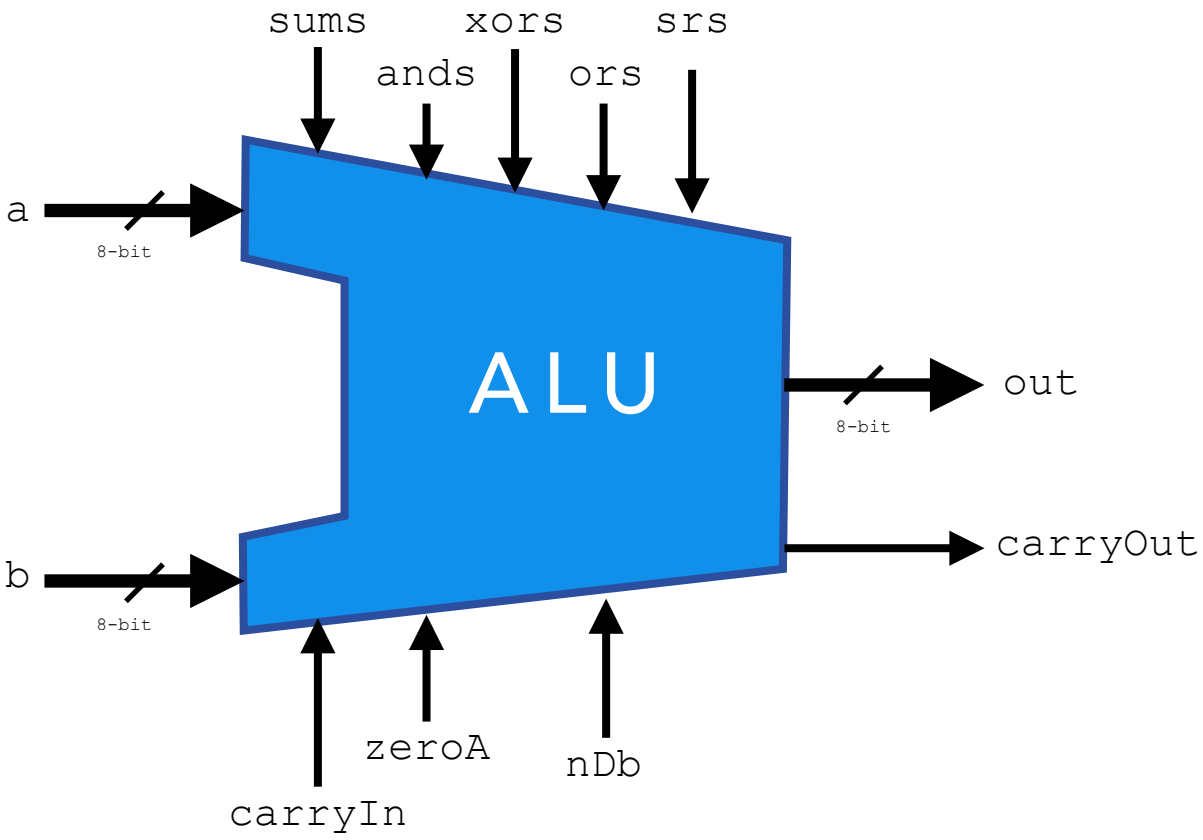
1 The 6502 CPU was very popular in the 1970s and 1980s and was the brain of many classic home computers, such as the Commodore 64 and the BBC Micro. It almost certainly influenced the design of the ARM CPU…

marks for the quality of your implementation as well — i.e. it will be looking to see whether you have made good reuse of the provided gates, the implementation does not contain unnecessary gates etc. The marks will then be combined to give a final mark out of 10, which will be your mark for this exercise. A mark of 40% (4/10) or above is a pass mark for the exercise.

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

Description of the ALU

In this coursework, you will implement our simplified 6502 *Arithmetic and Logic Unit*, or ALU[2]. A skeleton file `ALUcore.hdl` is provided as a starting point. The core of our simplified 6502 ALU is represented pictorially below and consists of two 8-bit inputs, `A` and `B`, and one 8-bit output. It also has a series of control inputs, which can be used to select what function the ALU performs and some outputs that are linked up to other sections of the CPU. These control inputs and outputs are described below.



As the diagram above shows, the `ALUcore` has two 8-bit inputs (`a` and `b`), one `carryIn` input, and seven control signals and produces one 8-bit output (`out`) and a carry output. The inputs and output are multibit, and so we represent them in the HDL as *buses*. A bus is just a collection of logic signals that we wish to keep together and sometimes treat as a whole (in this case they each represent a numeric value in binary). We can define them in the HDL by suffixing the pin name with

[2] You may find it helpful to read the chapter on 'Boolean Arithmetic' in the **nand2tetris** book (available online)for more information about designing an ALU, although the one we'll implement here is simpler.

square brackets, thus: `a[8]`. This denotes that `a` is a bus 8-bits wide, numbered `0-7`. Each individual bit can be accessed by placing the number of the required bit in the square brackets. So `a[3]` would access the fourth bit (remember, numbering starts from zero) from the right. Alternative, you can refer to a subset of a bus using the syntax `b[0..3]` or `a[4..7]`, where the numbers refer the to first and last bits in the range. However, note that you can only subset the output or input of a gate, you *cannot* subset the wires connecting gates together (this means you may sometimes need to define multiple wires coming form the same output). Again, re-read **Appendix A** of **nand2tetris** for more details on the HDL syntax.

Back to our ALU design then, two of the control signals, `zeroA` and `nDb`, modify the inputs `a` and `b` respectively *before they are used anywhere else in the ALU*. If `zeroA` is 1 (true), `a` should be replaced by all zeroes, otherwise if `zeroA` is 0 (false), `a` should be left unchanged. And, if `nDb` is 1, each bit of `b` should be inverted otherwise if `nDb` is 0, `b` should be left unchanged.

The other five control signals, `sums`, `ands`, `xors`, `ors`, and `srs`, are used to control the output produced by the ALU by selecting between the output of the five possible functions it can perform. The first four of these are used to select whether the two inputs are **Add**-ed, **And**-ed, **XOR**-ed, or **OR**-ed together respectively — each of these can be produced using combinations of the components you've created previous (e.g. `Add4C`, `And8`, etc.) and should connect the (*modified*, as above) `a` input bus of the ALU to the `a` bus of `And8` and the (*modified*, as above) `b` input bus to the `b` bus of `And8` (or `Or8` etc., as necessary). You can assume that only one signal will be active at once, and so the order you select between the possible outputs doesn't matter.

The `srs` signal is slight different, this should select that the output of the ALU, should be the a input shifted to the right (using the supplied `Shift8`), the `b` input should be ignored in this case.

The `carryIn` input of the `ALUcore` should be connected to the part of your implementation which adds the two values together (and any carry produced by an addition should be propagated out via `carryOut`, if no addition is being performed by the ALU then the `carryOut` should be set to 0 (false)).

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)

Hint Remember how hardware differs from software — in software we choose what we want the program to do next, in hardware, you produce all the possible outputs and then *select* the desired output…

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.

Supplied Gates

In the supplied git repository, you'll find a number of HDL files that conning implementations of various gates that you will need to use to implement the ALU described above. The supplied logic gates are listed below.

`Not8`	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.
`And8`	This has two input buses, `a` and `b`, and one output bus, `out`. Each bit of the output is the result of *anding* together the corresponding input bits in `a` and `b`.
`Or8`	This has two input buses, `a` and `b`, and one output bus, `out`. Each bit of the output is the result of *or-ing* together the corresponding input bits in `a` and `b`.
`Mux8`	This has two input buses, `a` and `b` and one output bus, `out`. Also present is a `sel` input, which is used to select whether input `a` or `b` is passed to out. If `sel` is false, input `a` is selected, otherwise input `b` is selected.
`Xor8`	This has two input buses, `a` and `b`, and one output bus, `out`. Each bit of the output is the result of *Xoring* together the corresponding input bits in `a` and `b`.
`Add4C`	This has two input buses, `a` and `b` and one output bus, `out`, along with a an additional `carryIn` input, and a `carryOut` output The output, `out`, is the result of adding together the two 4-bit numbers in `a` and `b` with the `carryIn` input. Any carry produced by the addition is available `carryOut` output to enable it to propagated to further gates.
`Shift8`	This has one input bus, `in`, and one output bus, `out`. This chip shifts each bit of the input precisely one position along to the right. So bit 1 of `in` becomes bit 0 of `out`, bit 2 of `in` becomes bit 1 of `out` etc. until finally bit 7 of `in` becomes bit 6 of `out`. Bit 7 of `out` is set to zero (i.e. `false`). **Note** since this is a *combinatorial logic* implementation its output is fixed, unlike the similar gate you created in lab exercise two, where the output changed over time.