ASSESSED INDIVIDUAL WORK

COMP1007
LARGE COURSEWORK 01

Steven R. Bagley

Submission Deadline:
14/12/2022 15:00

Version: 1.02

Throughout COMP1007, we've looked at how we can build a CPU from raw digital logic. In this coursework, we'll put some of the building blocks we have already built together to build some of the major building blocks that you might find in a simple CPU[1] using the **nand2tetris** hardware simulator and hardware description language.

This coursework is split into four sections. In the first section, you will create a series of combinatorial logic gates that will be used by the later parts of the systems. In the second section, you'll build a simplified version of the Z80 ALU.

In the third section, you will build some sequential logic gates that are used by the final section, where you will integrate the gates you have created in the previous sections to form a simplified section of the Z80 CPU. It is intended that in each section you will make use of the gates supplied, alongside any gates you create as part of the exercise, and the standard gates provided as part of **nand2tetris** (e.g `And`, `Or`, `Not`, `Mux`, `Bit`, etc.)

This document outlines how the coursework is assessed, and descriptions of the various logic circuits to be built, along with some tips on how to implement them.

This coursework is worth 20% of your final COMP1007 mark, and will be given a mark out of 40. As ever, a mark of 40% (i.e. a mark greater than 16) or above is a pass mark for this coursework.

The skeleton `.hdl` files and test scripts can be found in the following git repository:

`https://projects.cs.nott.ac.uk/2022-COMP1007/2022-COMP1007-LargeCoursework01`

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.

---

[1] This CPU is inspired by the classic 8-bit Z80 CPU. The Z80 CPU was very popular in the late-1970s and 1980s and was the brain of many classic home computers, such as the Sinclair ZX Spectrum, and the TRS-80.

01 Combinatorial Support Components

In this first section, we are going to build various combinatorial logic circuits that are used in parts of the CPU. If you have already completed the lab exercises then you will have built similar components already, so will be able to adapt them for this exercise. Some of them, however, are new and might require a bit of thought…

There are five support components that you need to implement. The five logic chips you are to implement are:

Gate	Description	Mark
`Mux4`	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` should be selected, otherwise input `b` should be selected. **Note:** You can assume that the single bit `Mux` is defined on the system already and takes three inputs, `a`, `b` and `sel`, and produces an output `out`.	2
`Mux8`	This is an 8-bit version of `Mux4`. It 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` should be selected, otherwise input `b` should be selected. **Note:** You can assume that the single bit `Mux` is defined on the system already and takes three inputs, `a`, `b` and `sel`, and produces an output `out`.	2
`Mux4Way8`	This is essentially the same a the `Mux8` except it can select between four different inputs (`a`, `b`, `c`, and `d`) and so `sel` is two bits wide — hence, `4Way`… **Hint** You should be able to build this using your implementation of `Mux8`	2
`Mux8Way8`	This is essentially the same as the `Mux4Way8` defined above, however this time there are eight inputs and `sel` is three bits wide. You should adapt your earlier implementation to reflect this change.	2

Gate	Description	Mark
`HiLoMux`	This has one *8-bit* input bus, `in`, and one *4-bit* output bus, `out`. Also present is a `sel` input, which is used to select what appears on out. If `sel` is false, then `out` should contain the lower 4-bits of `in` (i.e. `in[0]`, `in[1]`, `in[2]`, `in[3]`). If `sel` is true, then `out` should contain the upper 4-bits of `in` (i.e. `in[4]` mapped to `out[0]`, `in[5]`, mapped to `out[1]`, etc.). In other words, the `HiLoMux` can be used to *select* a nibble from a byte… We'll use this circuit in a later exercise to enable the 4-bit `ALUcore` below to do 8-bit operations.	2

Note only `Mux4` and `HiLoMux` are required to complete later parts of this coursework.

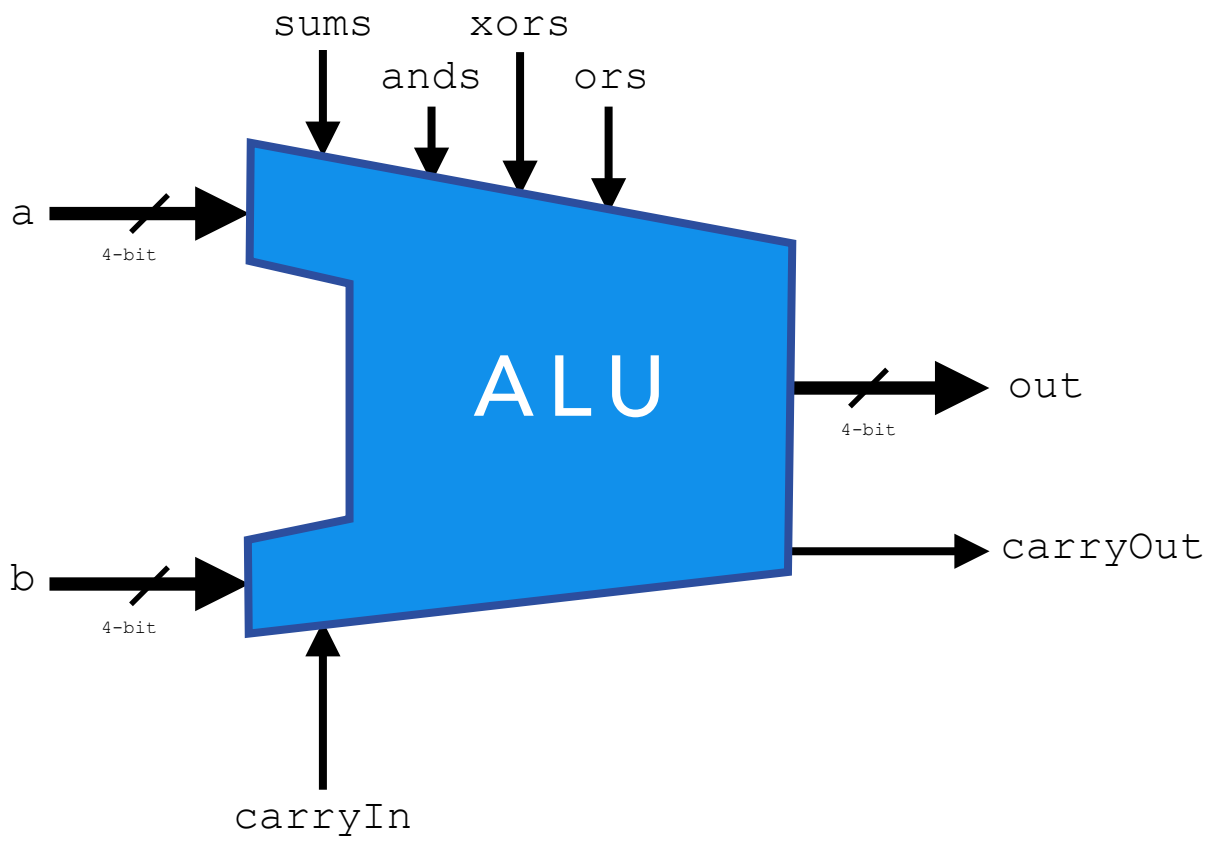
02 Arithmetic and Logic Unit

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 could add 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 Z80 microproccessor, and is much simpler to understand than the **nand2tetris** ALU we considered in the online session on the 26th 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 (Details of the supplied gates can be found in Appendix A).

In this section, you will implement our simplified Z80 *Arithmetic and Logic Unit*, or ALU[2]. A skeleton file `ALUcore.hdl` is provided as a starting point. The core of our simplified Z80 ALU is represented pictorially below and consists of two 4-bit inputs, `A` and `B`, a 4-bit output and carry inputs and outputs. Even though this ALU is only 4-bits wide, it is still able to perform 8-bit computations as we will see in section four.

The ALU 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.

[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.



The inputs `a` and `b` along with the output, `out`, are multi-bit, and so we represent them in the HDL as *buses*. A bus is just a collection of logic signals that we keep together (in this case they reach represent a numeric value in binary). We can define them in the HDL by suffixing the pin name with square brackets, thus: `a[4]`. This denotes that `a` is a bus 4-bits wide, numbered `0-3`. 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. Again, re-read the relevant sections of **nand2tetris** for more details on the HDL syntax.

The four control signals, `sums`, `ands`, `xors`, and `ors`, are used to control the output produced by the ALU by selecting between the output of the four possible functions it can perform. These are used to *select*[3] whether the two inputs are **Add**-ed, **And**-ed, **XOR**-ed, or **OR**-ed together respectively — each of these can be produced using the components supplied. So the output from the addition should *only* be selected if `sums` is true, the output from the logical and should only be selected if `ands` is true, and so on. If none of these signals are true, then the ALU should produce no output (i.e. zero, or `false`).

You can assume that only one control signal will be active at a time, and so the order you select between the possible outputs doesn't matter. The `carryIn` input of the `ALUcore` should be connected to the `carryIn` of your `Add4C`, while the `carryOut` signal of the `ALUcore` should be connected to the `carryOut` of your `Add4C` when `sums` is true, otherwise it should be false.

[3] Remember this is hardware, so we perform all the operations and select the desired output…

This completes the implementation of our ALU's core — in the final section, you will build some more logic to drive the core, enabling it to perform subtraction as well as addition, and also to be able to perform 8-bit operations.

Assessment Criteria

The gitlab pipeline will mark your program and give you feedback on how well it works. Specifically, 10 marks are awarded for this iteration as follows:

- 2 marks are available if `ALUcore` can successfully add two values together
- 2 marks are available if `ALUcore` can successfully and two values together
- 2 marks are available if `ALUcore` can successfully or two values together
- 2 marks are available if `ALUcore` can successfully xor two values together
- 2 marks are available if `ALUcore` can successfully peform all operations…

Various test scripts are provided for the `ALUcore` (`ALUcore-add.tst`, `ALUcore-and.tst`, etc.) which can be used to test each of these individual cases

Successfully completing parts one and two will give you a passing grade for this exercise…

03 Sequential Support Components

The components you will build in this section are similar to `Register8` in lab exercise two and you can assume the `Bit` logic gate is available for you to use. Chapter 3 of the **nand2tetris** book also contains detailed descriptions of some related circuits.

Gate	Description	Mark
`Register4`	This has one 4-bit input bus, `in`, and one output bus, `out`, and is designed to store a single *byte* of information. As with `Bit`, a further input `load` controls whether the output should be updated to reflect the new input value (as `Bit`).	2
`RegisterHiLo8`	This has one 4-bit input bus, `in`, and one 8-bit output bus, `out`, and is designed to store single byte of information. Unlike `Register8`, `RegisterHiLo8` only allows 4-bits of the value to be loaded at a time — so updating all 8-bits requires two separate loads. Therefore, `RegisterHiLo8`, has two further inputs `loadLo`, and `loadHi` to select whether the lower 4-bits should be stored (bits 0–3), or the upper 4-bits (bits 4–7) should be updated to reflect the input value, `in`, respectively. **Hint** The key to this is to store the value as two separate 4-bit value until you generate the output, `out`, of the circuit.	3

04 ALU Data Path

The Z80 CPU is an 8-bit CPU but it was implemented using a 4-bit ALU. This meant that it had to perform 8-bit maths as a sequence of two operations first on the low 4-bits of the data, then on the high 4-bits of the data, propagating any carry as necessary. In this section, you are going to implement similar logic.

The three gates in this section wrap up a **single** 4-bit `ALUcore` that you implemented in the second section and provides the additional logic needed to enable it to perform 8-bit operations. Operationally, these gates will work as if they had two `ALUcore`s connected in parallel with the `carryOut` of the first linked to the `carryIn` of the other but, instead of using two `ALUcore`s, `ALUDataPath` will perform the selected operation as a sequence of two operations one after the other **using a single `ALUCore`**: first, on the lower 4-bits, or nibble[4], of the 8-bit values, and then secondly, perform the same operation on the high nibble of the 8-bit value — preserving any carry value from the first operation and feeding back into the second operation.

There are three gates to implement for this section. The first gate, `ALUChopper`, takes in 8-bit values, stores them and enables us to chop them into the 4-bit nibbles necessary to perform the calculation. The second gate, `ALUProcess`, then uses the two 4-bit nibbles from the first gate to perform an 8-bit calculation using a single 4-bit `ALUcore` and produce an 8-bit output. Finally, the last gate `ALUDataPath`, just wraps up the first two into an easy to use package.

Gate	Summary	Mark
`ALUChopper`	Stores the input for the ALU and chops them into nibbles.	3
`ALUProcess`	Performs an 8-bit computation as a sequence of operation using the four 4-bit nibbles produced by `ALUChopper`	5
`ALUDatapath`	Wraps up the two gates into a usable component	2

ALUChopper

This logic gate is used to both *store* the two 8-bit values required for the computation by the ALU and also to chop the stored values into two 4-bit values that can then be fed into the ALU. This gate only has a single 8-bit input, `in` which is used to provide the 8-bit input values. The values for the ALU will be provided in sequence, and so two further inputs `aLoad` and `bLoad` are used to select whether the value on `in` should be stored ready to be used for the `a` input or `b` input to the `ALUcore`.

The logic gate has two 4-bit outputs, `aHiLo` (which should contain the value stored when `aLoad` is true) and `bHilLo` (which should contain the value stored when `bLoad` is true) which are used to provide 4-bit outputs ready for `ALUProcess`. To select whether these contain the the high nibble, or the low nibble of the output there is a final input `hiLo` which switches between them. If `hiLo` is

[4] 4-bit binary values are referred to as a *nibble* — because they are half a byte… The lower nibble, would there for be be bits 0-3 of a byte, and the higher nibble be bits 4-7.

true, then aHiLo and bHiLo contain the upper nibble, otherwise if hiLo is false, they contain the lower nibble.

ALUProcess

ALUProcess wraps up a **single** 4-bit ALUcore, and enables it to perform 8-bit computations (sequentially). This is done by using that ALUcore to compute the lower nibble (bits 0—3) first, and storing the result in part of a register, and then performing the same calculation on the high nibble (bits 4—7) and storing that part of the computation in the same register. For boolean operations (And, Or, Xor), this alone will produce the correct result but for addition we need to go one step further since we need to ensure that the carry produced by the first half of the computation is stored and then fed back into the ALUcore (via carryIn) for the second half of the computation). In addition, we will also need to store the carryOut bit when we update high nibble of the result to ensure it updates the output at the same time as the rest of the computation.

Several of the inputs to this chip are familiar: sums, ands, xors, and ors are used to select the operation that the ALUcore will perform and so can be wired directly to the equivalent inputs on the single ALUcore in your solution. The inputs a and b are used to provide the input to ALUcore (they'll be connected to the ALUChopper above to select the correct bits at the correct time), and again can be connected directly to ALUcore.

The other inputs and outputs need a little more explanation. The input hiLo is straight-forward and, as with ALUChopper, is used to select whether ALUcore is working on the high (hiLo=true) or low (hiLo=false) nibble of the computation. You will need to use this (in combination with some of the other inputs to ensure the correct part of the chip operates).

The output, out, contains the result of the calculation performed by the ALUcore. However, since ALUcore will need to perform two sequential operations to build up the result, you will need to use a register to store the high and low nibbles of the result as they are calculated separately… The input resLoad is used to tell this register when to store the output of the ALUcore and should be combined with hiLo to select which nibble is updated. When the relevant operation is performed on the high nibble, the carryOut bit from the ALUcore should also be stored before being presented to the ALUDataPath's carryOut.

Hint: Remember, you'll need to preserve the carryOut *Bit* from the ALUcore when it performs the operation on the lower nibble.

ALUDataPath

The final gate, ALUDataPath, ties the other two together to form a usable system. Most of its inputs can be directly connected to the equivalent input on ALUChopper and/or ALUProcess. The inputs op1Load and op2Load are used to specify whether dataIn contains the first or second operand for the computation and so can be used to drive aLoad and bLoad respectively

on the `ALUChopper`. In addition, the `aHiLo` output of `ALUChopper` can be connected to the `a` input of `ALUProcess`.

The final input, `notOp2`, is necessary to enable the ALU to perform a subtraction. It used to select whether the second input (`b`) to `ALUProcess` is inverted (i.e. fed through a `Not4`) or not. If `notOp2` is true, then `ALUProcess` should be fed the inverted version. By inverting the second operand, and setting `carryIn` to be 1, we can get the ALU to subtract by adding the negative version of a number (remember, the two's complement representation is formed inverting and adding one…).

Once again, a test script is provided in the repository to enable you to ensure that your implementation is working correctly. However, it'll probably worth trying the circuit out 'by hand' so you get an idea how it works. Appendix B describes how the ALUDataPath can be driven to perform a calculation.

DataPathDriver

The pipeline will also award an additional five marks if your `ALUDataPath` works under the control of the supplied `DataPathDriver` to correctly perform calculations (you can test this yourself by running the provided `DataPathDriver.tst` test script in **nand2tetris**).

APPENDIX A

Several logic gates are provided for you as part of the git repository and it is expected that you will make use of these in your solution. Details of the gates provided are given below. In addition, should you need to, you can make use of any of the gates we have already seen, such as `And`, `Or`, `Xor`, `Not`, `Mux`, `Bit`, `FullAdder`, `HalfAdder`, etc.

`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`.
`Add4C`	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.
`Register8`	This gate has one 8-bit input bus, `in`, and one output bus, `out`, and is designed to store a single *byte* (8-bits) of information. As with pre-supplied `Bit`, a further input `load` controls whether the output should be updated to reflect the new input value (when `true`), or should preserve the output.

APPENDIX B

Understanding the Datapath

Once you've built a working datapath, it can be instructive to manually 'waggle' (i.e set them to a specific value) the inputs to it to see how the data moves between the various parts of datapath and a calculation is performed. It is helpful to look at the 'Internal Pins' section of your datapath when doing this to see the value progress to the various section.

If you waggle the pins in the following sequence, you should find the system adds the values of 42 and 23 together (unless otherwise stated, ensure all other pins are at zero):

- Set `dataIn` to 42
- Set `carryIn` to 0
- Set `sums` to 1
- Set `op1Load` to 1
- Press the Hardware Simulator clock button twice — this should cause the value of 42 to be stored inside your `ALUcore` (you should see it appear in the internal pins section). The first number is loaded, so we can now load the second.
- Set `dataIn` to 23
- Leave `carryIn` at 0 and `sums` at 1
- Set `op1Load` to 0
- Set `op2Load` to 1
- Press the Hardware Simulator clock button twice — this should cause the value of 23 to be stored inside your `ALUcore` (you should see it appear in the internal pins section, along with the 42 you already stored). Both numbers are now loaded, so we can start performing the calculation.
- Set `op1Load` and `op2Load` to both to 0
- Leave `carryIn` at 0 and `sums` at 1
- Ensure `hiLo` is at 0
- Set `resLoad` to 1
- Press the Hardware Simulator clock button twice — this should cause the lower nibbles to be added together. You should see part of `dataOut` be updated (probably with the value of 1 — although it may be different if you have performed previous calculations).

- Set `hiLo` to 1 and ensure `resLoad` is 1
- Press the Hardware Simulator clock button twice — this should cause the higher nibbles to be added together. You should see `dataOut` be updated with the result, 65.
- Set `resLoad` to 0 since we have finished the calculation.

To perform any operation, we need to move through the same set of four states, setting the relevant control inputs to `ALUDataPath`... In the first state, we load the first operand into the `ALUDataPath`. In the second state, we load the second operand into the `ALUDataPath`. In the third state, we calculate the result of the lower nibble and in the fourth and final state we calculate the high nibble of the result.

Why not experiment to see how you would need to modify the process above to calculate 42 - 23 rather than 42 +23, or how to perform a boolean operation.