# G51CSF LAB EXERCISES
# ASSESSED EXERCISE #3A

Steven R. Bagley

Submission Deadline:
**05/12/2017**

## Introduction

In the first G51CSF assessed exercise, you will implement heavily simplified versions of components inside the Z80 CPU[1]. Each of the weekly exercises get you to build different components of the CPU, although some of the more complex components will be provided for you. In total, this coursework is **worth 30%** of your final G51CSF mark.

In the first two assessed exercises, you built all the components needed to build a (heavily) simplified version of the data path driving the ALU of the Z80, and in this third exercise, we'll put those components together to implement and drive that datapath.

Please note that this exercise has been split into two parts 3A and 3B, to give you extra time to complete them. Exercise 3B will be made available next week, and will be due in on the 12/12/2017. The logic circuits in exercise 3A and 3B are worth double points, i.e. 4 marks each.

## Essential Setup

There is only one component to build this week, although it is somewhat involved, so I strongly recommend reading this document carefully. Both of them require you to reuse some of the gates you created in previous weeks. Therefore, once you have checked out the skeleton files from git, **you will need to copy several of the** `.hdl` files you created in exercises one and two into the extracted folder. Don't forget to use `git add` to add these into your project!

From exercise one, you will need to copy `Add4.hdl`, `ALUcore.hdl`, `And4.hdl`, `Mux4.hdl`, `Not4.hdl`, `Or4.hdl`, `HiLoMux.hdl` and `Xor4.hdl` into the extracted folder, and from exercise two you will need copy over your implementations of `Bit.hdl`, `Register4.hdl`, `Register8.hdl`., and `RegisterHiLo8.hdl`. If you created any extra gates used in your implementation, then remember to copy them over as well.

Do not worry if you did not manage to complete the previous exercises, or your implementation doesn't work correctly since implementations (implemented in Java rather than HDL) of all the previous gates required for this exercise will be made available after the hand-in date for first assessed exercise. These Java versions can be loaded as plug-ins into the Hardware Simulator.

In addition, a Java version of the `ALUDataPath` (below) will be made available along with Exercise3B.

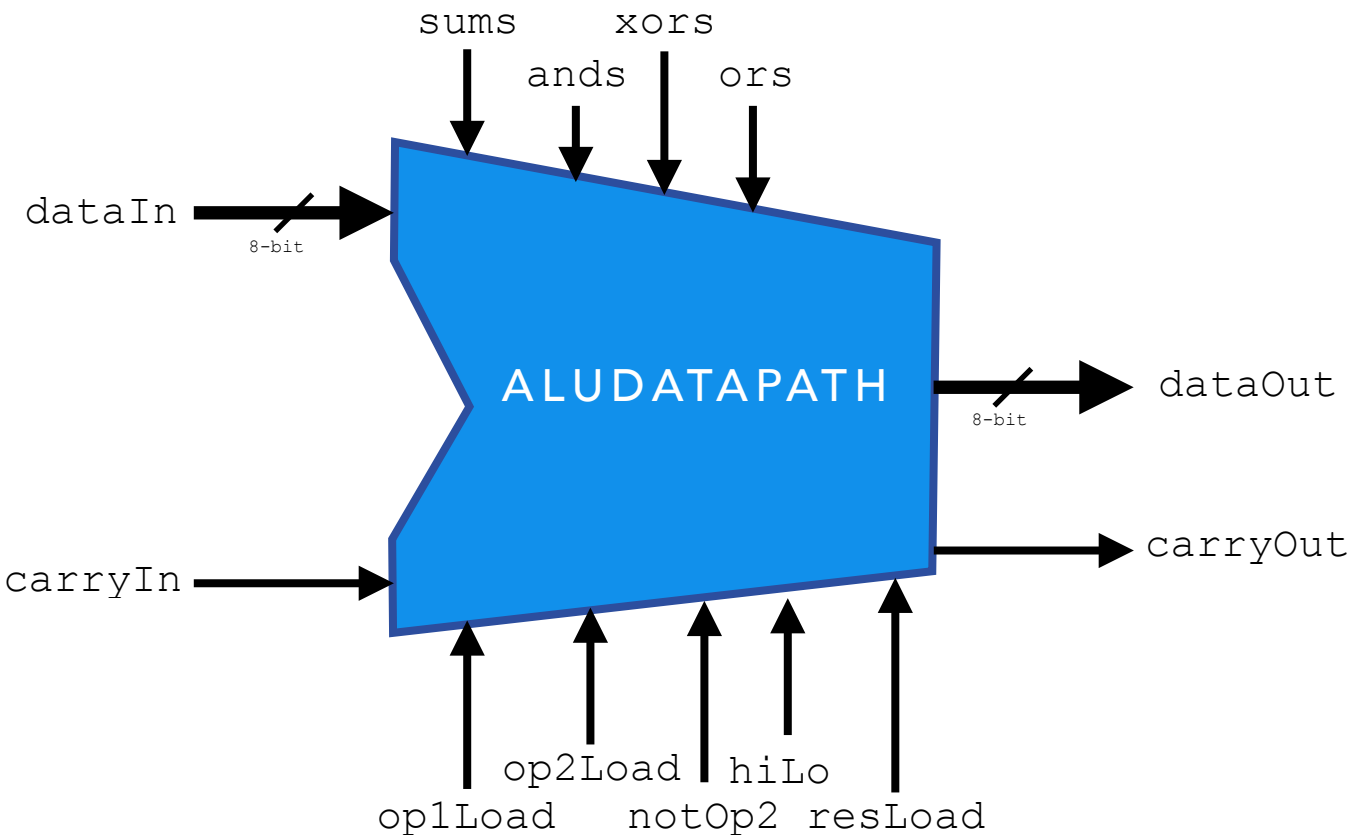
---

1 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.

## ALUDataPath

The gate to build this week is `ALUDataPath`. This wraps up **a single** 4-bit `ALUcore` that you implemented in the first assessed exercise and provides the additional logic needed to enable it to perform 8-bit operations. Operationally, this circuit works as if it had two `ALUcores` connected in parallel with the `carryOut` of the first linked to the `carryIn` of the other but, instead of using two `ALUcores`, `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[2], 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.

The diagram below shows the input and outputs to `ALUDataPath`. Several 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 input `hiLo` is also straight-forward and is used to select whether the `ALUcore` is fed by the high (`hiLo`=true) or low (`hiLo`=false) nibble of the inputs.




The input to this circuit is a single 8-bit input, `dataIn`, along with a single `carryIn` input. This single `dataIn` input is used to provide both inputs to our `ALUcore` (`a` and `b`), with the two values being loaded in sequence. Two additional inputs, `op1Load` and `op2Load`, define whether `dataIn` contains the input for `a` (`op1Load` is true), or b (`op2Load` is true). Note, that since the values are

[2] 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.

loaded in sequence, the two values must be stored in registers before the selected nibble from each input is presented to the ALU.

The output, `dataOut`, 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 and store the high and low nibbles of the result 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`.

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

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

Once again, a test script is provided in the download 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.

## 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. In exercise 3B, we will develop a simple state machine that will move through these four states and generate the necessary signals for ALUDataPath to perform the calculations — in the same manner that the control path of a CPU does.

Until then, 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.