G51CSF LAB EXERCISES
ASSESSED EXERCISE #3B

Steven R. Bagley

Submission Deadline:
12/12/2017

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. The logic circuits in exercises 3A and 3B are worth double points, i.e. 4 marks each.

Essential Setup

There are several components to build this week, and I strongly recommend reading this document carefully. They require you to reuse some of the gates you created in previous weeks, including the ALUDatapath from exercise 3A. 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!

As with Exercise 3A, 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, from exercise two you will need copy over your implementations of `Bit.hdl`, `Register4.hdl`, `Register8.hdl`., and `RegisterHiLo8.hdl`. In addition, you'll need to copy over your `Mux8` gate from Exercise 2, and from exercise 3A, you'll need your `ALUDataPath`. 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.

Driving the Datapath

In this exercise, we are going to generate sequential logic components that will drive the `ALUDataPath`, you built in exercise 3A to either add two numbers together or subtract one number from the other. Effectively, we'll be automating, in hardware, the process described in the 'Understanding the Datapath' section of exercise 3A.

---

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.

There are two parts that you will build — the first part is a state machine that will contain four states, that it'll move through one by one, i.e. it'll start off in state zero, then move to state one, to state two, to state three and then back to state zero again. In other words, the next state does not depend on any input value and is purely based on the value of the current state.

You will then use this state machine in the second logic circuit to automatically drive the `ALUDataPath`'s inputs in the right order to add or subtract the two values.

Note it is worthwhile familiarising yourself with how the ALUDataPath works by manual controlling its inputs as described in the pins in the 'Understanding the Datapath' section of exercise 3A.

StateMachine

As described above this gate should count through the states 0,1,2 and 3 in order. The gate has one output, `state`, which outputs the two-bit value of the current state. The gate has an input, `reset` which forces the current state to be zero. When reset is 1, the output, `state`, should be zero, then when reset goes back to zero, the state should start counting again (i.e. move to state 1).

To build this you will need to store the current state (two bits) — it probably makes sense to use `DFF` directly here, rather than using `Bit`. You'll also need to provide the logic to take the current state and generate the next state (i.e. the current state *plus* one) to feed back to the input of your `DFF` as the next state. This logic will also need to handle processing the reset.

DataPathDriver

This gate has two inputs, `a` and `b`, which contain the two numbers to be added, or subtracted (which operation is performed is decided by the third input, `subtract` — when `subtract` is true, the gate should calculate `a - b`, otherwise it should calculate `a + b` by using the `ALUDataPath`). The `reset` input should be connected directly to your `StateMachine` gate — which will sequence the order of events in this gate. There is also a single output, `sum`, which should be connected to the `dataOut` of the `ALUDataPath`.

Although this is a sequential component, building it can be done just using combinatorial logic based on the `state` output of the `StateMachine`, and the inputs. The following table lists what your circuit should for each state. Unless otherwise stated, the other control inputs to `ALUDataPath` should be false.

`state`	Description
0	In this state, we need to load the value of a into the `ALUDataPath`. Your logic should there for ensure that `a` is connected (i.e. switched) to the `dataIn` input of your `ALUDataPath`. You'll also need to ensure that `ALUDataPath`'s `op1Load` is true, and that both `op2Load` and `resLoad` are false during this state

state	Description
1	In this state, we need to load the value of `b` into the `ALUDataPath`. Your logic should there for ensure that `b` is connected (i.e. switched) to the `dataIn` input of your `ALUDataPath`. You'll also need to ensure that `ALUDataPath`'s `op2Load` is true and that both `op1Load` and `resLoad` are false during this state
2	In this state, we need to start calculating the result. Therefore, `resLoad` should be true, and `op1Load` and `op2Load` should both be false. Here, we are calculating the lower part of the result so `hiLo` should also be false.
3	In this state, we need to finish calculating the result. Therefore, `resLoad` should be true, and `op1Load` and `op2Load` should both be false. Here, we are calculating the upper part of the result so `hiLo` should also be true.

Since this driver, can only add or subtract, it is safe to leave `sums` true all the time. However, you'll need to set the value of notOp2 and carryIn accordingly to perform a subtraction (see explanation in exercise 3A for more info).

Once you have written your gate, you can test it by setting the inputs `a` and `b` with some values. and then pressing the clock button in the simulator 8 times (i.e. 4 tick-tock stages of the clock, one for each state) you should fine that the result in sums matches the calculation.

WARNING I've noticed the Hardware Simulator can sometimes produce some weird results when executing this more than once. Therefore, I recommend reloading the chip before performing a test.