G51CSF LAB EXERCISES
ASSESSED EXERCISE #2

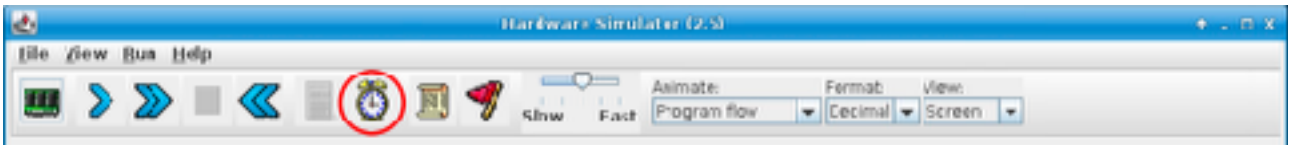
Steven R. Bagley

Introduction

In the first G51CSF coursework, you will implement a heavily simplified version of the 6502 CPU[1]. Each of the weekly exercises will get you to build different sections 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.

The first set of exercises used *combinatorial logic* to construct many of the various logic circuits that form the basis of a CPU. With this exercise, we are going to start producing some of the various *sequential circuits* used in a computer (both inside the CPU, and outside to form memory). As you should recall from the lecture, sequential logic circuits are those that can store information and who's output is based not just on some combination of the inputs but also on the previous state of the circuit.

The **nand2tetris** Hardware Simulator lets us construct this kind of circuit by using the supplied `DFF` gate (which mimics a D flip-flop), and the simulated clock. This clock can either be driven by a test script, or by pressing the 'clock' toolbar button (highlighted):



The clock button enables you to simulate the oscillations of the clock signal within a computer, producing both a *tick* and a *tock* phase. The `DFF` gate is setup to update its output pin `out` on the *tock* phase of the clock, storing whatever is currently at its input pin `in`.

Do not worry if you didn't manage to complete the previous exercise, or your implementation doesn't work correctly since **nand2tetris** provides built-in implementations (implemented in Java) of all the gates required. To ensure you use these built-in implementations, we recommend you keep the files for this exercise in **a separate directory** as they were supplied in the download from Moodle.

You will almost certainly find it impossible to complete these exercises unless you have read section A.7 — Sequential Chips — of Appendix A of *'The Elements of Computing Systems: Building a Modern Computer From First Principles'* — the book which accompanies nand2tetris — when implementing these circuits. You can either purchase the book via Amazon or find the relevant chapters on their website at: `http://nand2tetris.org/course.php`

---

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 that

Sequential Components

The components you will build in this exercise are similar to those in the third **nand2tetris** project and you will need to implement the logic circuits using the `DFF` primitive gate, alongside the gates you implemented in the previous exercises.

We suggest that you work on them in the order specified: `Bit`, `Register`, `RAM8`, `RAM64` `RAM512`, `RAM4K`, and `RAM16K`, since you can then use the earlier gates to build the more complex ones (e.g. `Register` can be defined relatively simply in terms of `Bit`, rather than having to specify it all from scratch). Again, Chapter 3 of the **nand2tetris** book contains more detailed descriptions of each circuit.

For this exercise, you will be awarded a grade for each of the gates: `Bit`, `Register`, `RAM8`, and `RAM64`, along with a single grade for completing *all* the gates: `RAM512`, `RAM4K`, `RAM16K`. This is because each of the RAM gates feature near identical implementations[2].

Hint: You will be required to create feedback loop to implement `Bit`, where the output of a `DFF` is connected back to its input (possibly via some combinatorial logic). This can lead to errors from the Hardware Simulator if you use the name of `Bit`'s output pin (`out`) as an input to another gate. You can get around this by specifying two pins connect to the out pin of the `DFF`, one that connects to output pin of the `Bit`, and another that you can use to connect to other parts of the chip, e.g.:

```
DFF(in=…, out=out, out=x);
```

where `x` can be used to connect to other logic gates.

Download and extract the `.ZIP` file associated with this coursework. Inside it you will find skeleton `.hdl` files and test scripts for each of the components you need to implement.

With all these chips, you should build the later chips, by re-using the previous chips you have created (i.e. build `RAM8` up using `Register` , `RAM64` in terms of `RAM8` etc.). The ten sequential logic chips you need to implement are:

[2] This may be a hint about how they should be implemented…

`Mux8Way8`	This is essentially the same a the `Mux8Way16` defined in the first lab exercise, however this time the bus widths are 8-bit in size, rather than 16-bit. You should adapt your earlier implementation to reflect this change. **Note** You will need to copy your completed `Mux8.hdl` file from the first exercise into the same directory as `Mux8Way8.hdl` for you to be able to use your `Mux8`…
`Mux4Way8`	This is essentially the same a the `Mux4Way16` defined in the first lab exercise, however this time the bus widths are 8-bit in size, rather than 16-bit. You should adapt your earlier implementation to reflect this change. **Note** You will need to copy your completed `Mux8.hdl` file from the first exercise into the same directory as `Mux4Way8.hdl` for you to be able to use your `Mux8`…
`Bit`	This has one input, `in`, and one output, `out`, and is designed to store a single bit of information. A further input `load` controls whether the output changes to match the input, or freezes at its current value. If `load` is true (1), then the output should be updated to match the input (i.e `out(t) = in(t-1)`), otherwise, if `load` is false, the output should remain unchanged (i.e `out(t) = out(t-1)`)
`Register`	This has one 8-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`).
`RAM8`	This has one input bus, `in`, and one output bus, `out`. It defines an array of 8 8-bit registers. A second input bus, `address`, indexes which register is to be accessed. The `out` bus should contain the value stored in the register specified by address. A further pin `load` is used to say whether the register specified by address should be updated by the value on the bus, `in`.
`RAM64`	As `RAM8` but this time defining 64 memory locations (i.e. 64 registers) indexable by `address`.
`RAM512`	As `RAM64` but this time defining 512 memory locations (i.e. 512 registers) indexable by `address`.
`RAM4K`	As `RAM512` but this time defining 4096 memory locations (i.e. 4096 registers) indexable by `address`.
`RAM16K`	As `RAM4K` but this time defining 16384 memory locations (i.e. 16384 registers) indexable by `address`.

Bonus unassessed task

If you manage to finish the above exercise quickly, you might want to try combining some of the circuits you created this week with those you created last week to produce some working circuits.. You will need to **copy** the `.hdl` files for the various chips you use into a separate directory to enable them to work together.

I suggest trying to create logic chip that has one input bus, `in`, which is used to specify an 8-bit number. It should also have one output bus, `total` which keeps a running total of all the numbers that are fed into the input bus, in. Finally, there should be an `add` pin that, when true, signifies a new value should be added to the total and a `reset` pin which when true resets the value of total to `zero`.

There's no test script for this — you'll need to use the clock button on the toolbar to manually test it yourself.

Remember *this bonus task is unassessed, and just for fun* — but you'll probably learn a lot about how to wire things up[3]…

[3] And if you are really keen, why not try making it subtract the values from a specific total (such as 501) rather than add them one…