

G51CSF LAB EXERCISES

ASSESSED EXERCISE #2

Submission Deadline:

28/11/2017

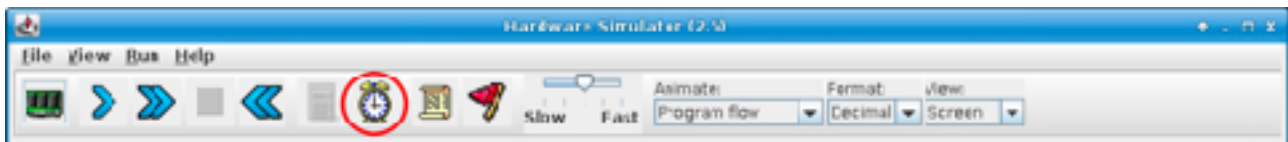
Steven R. Bagley

Introduction

In the first G51CSF assessed exercise, you will implement heavily simplified versions of components inside the Z80 CPU¹. 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.

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 whose 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 when checked out from gitlab.

Don't forget to test the chips you design, both by hand to ensure that they do what you expect and also by using the test scripts provided.

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.

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>

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`, `Register4`, `Register8`, `RegisterHiLo8`, etc. since you can then use the earlier gates to build the more complex ones (e.g. `Register4` and `Register8` 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 detailed descriptions of some related circuits.

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 `Register8`, `RAM64` in terms of `RAM8` etc.). The nine sequential logic chips you need to implement are listed below.

The first four logic chips will be used in building our implementation of a CPU similar to the Z80, while the last five demonstrate one mechanism for building RAM chips out of logic gates.

<code>Bit</code>	This has one input, <code>in</code> , and one output, <code>out</code> , and is designed to store a single bit of information. A further input <code>load</code> controls whether the output changes to match the input, or freezes at its current value. If <code>load</code> is true (1), then the output should be updated to match the input (i.e $out(t) = in(t-1)$), otherwise, if <code>load</code> is false, the output should remain unchanged (i.e $out(t) = out(t-1)$)
<code>Register4</code>	This has one 4-bit input bus, <code>in</code> , and one output bus, <code>out</code> , and is designed to store a single byte of information. As with <code>Bit</code> , a further input <code>load</code> controls whether the output should be updated to reflect the new input value (as <code>Bit</code>).

Register8	This has one 8-bit input bus, <code>in</code> , and one output bus, <code>out</code> , and is designed to store a single byte of information. As with <code>Bit</code> , a further input <code>load</code> controls whether the output should be updated to reflect the new input value (as <code>Bit</code>).
RegisterHiLo8	<p>This has one 4-bit input bus, <code>in</code>, and one 8-bit output bus, <code>out</code>, and is designed to store single byte of information. Unlike <code>Register8</code>, <code>RegisterHiLo8</code> only allows 4-bits of the value to be loaded at a time — so updating all 8-bits will take two separate loads. Therefore, <code>RegisterHiLo8</code>, has two further inputs <code>loadLo</code>, and <code>loadHi</code> 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, <code>in</code>, respectively.</p> <p>Hint The key to this is to store the value as two separate 4-bit value until you generate the output, <code>out</code>, of the circuit.</p>
Mux8	<p>This is an 8-bit version of <code>Mux4</code> that you built last week. It has two input buses, <code>a</code> and <code>b</code> and one output bus, <code>out</code>. Also present is a <code>sel</code> input, which is used to select whether input <code>a</code> or <code>b</code> is passed to <code>out</code>. If <code>sel</code> is false, input <code>a</code> should be selected, otherwise input <code>b</code> should be selected.</p> <p>Note: You can assume that the single bit <code>Mux</code> is defined on the system already and takes three inputs, <code>a</code>, <code>b</code> and <code>sel</code>, and produces an output <code>out</code>.</p>
Mux8Way8	This is essentially the same as the <code>Mux8Way16</code> 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.
Mux4Way8	This is essentially the same as the <code>Mux4Way16</code> 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.
RAM8	<p>This has one input bus, <code>in</code>, and one output bus, <code>out</code>. It defines an array of 8 8-bit registers. A second input bus, <code>address</code>, indexes which register is to be accessed. The <code>out</code> bus should contain the value stored in the register specified by <code>address</code>. A further pin <code>load</code> is used to say whether the register specified by <code>address</code> should be updated by the value on the bus, <code>in</code>.</p> <p>Note: You can assume that the chip <code>DMux8Way</code> from the first lab exercise is pre-defined and available for use on the system.</p>
RAM64	As <code>RAM8</code> but this time defining 64 memory locations (i.e. 64 registers) indexable by <code>address</code> .

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 up²...

² 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 on...