Week 5 assignment: Hack assembly practice (part 2)

1 Tasks

1. Learn about proper input polling.
2. Learn about left and right shifts and use them to investigate the Collatz conjecture and implement a faster multiplication algorithm.
3. Implement a rudimentary control scheme for a game.

2 Required software

For this lab, you will need the assembler and CPU emulator from the nand2tetris software suite. The demonstrations in the video lectures should give you a good idea of how to use this software, and the software and documentation are both available from the unit page here. All of it runs on Windows, Linux and Mac OS.

In order to run this software (and anything else from the nand2tetris suite) on your home computer, you will need to install version 8 of the Java Runtime Environment, which you can download here. (It's already installed on the lab computers.) If you are getting an error about javaw.exe being missing, the most likely reason is that you don't have the Java Runtime Environment installed.

You will eventually be able to run our own fork of the nand2tetris suite on lab computers by entering in a terminal "module load nand2tetris" followed by "CPUEmulator.sh", without the need to download external files. For now, this still loads the original version of the emulator rather than our fork — we will update this sheet when that changes.

3 The missing step in handling input

Load up `checker2.asm` from part 1. That program toggled between displaying two patterns on the screen depending on whether or not the "c" was held. Adapt this into a new program `checker3.asm` which instead toggles between the two patterns each time the user presses the "c" key.

Codestuff.online link: Coming soon!

Your first attempt at this will probably break! Here's why. The most natural approach here is to take your infinite screen-drawing loop from `checker2.asm`, add a variable `pattern` controlling which pattern to draw, and toggle `pattern` whenever the "c" key is pressed. The problem is that this loop will run hundreds of times a second, and the average keypress duration is around 75ms — roughly one thirteenth of a second. So as you run through your loop, each time "c" is pressed, you end up toggling `pattern` so many times that the final value is effectively random.

The situation is analogous to level-triggering versus edge-triggering in the first part of the unit. When you're polling for a keystroke, you often don't want something to trigger based on whether the key is currently pressed (by level) but instead based on whether the key has been pressed or released since the last time you checked (by edge). Here an iteration of your `pattern`-updating loop is analogous to a clock cycle. To solve this exercise successfully, you'll need to work out a way of making the toggle effect edge-triggered.

4 Rotation and shifting

Besides masking from part 1, there's one more common type of bitwise operation: shifting. When we apply a *left shift* operation to a word x, every bit in x is shifted one place to the left, with a zero being added to the right to keep the length the same. For example,

```
         1111111001111111                1010111000011011
becomes  1111110011111110,       becomes 0101110000110110.
```

This is actually equivalent to multiplying x by two — can you see why?

Any modern programming language (including assembly languages!) have native support for shift operations. In C, `x << a` applies the left shift operation to `x` a total of `a` times. Write a Hack assembly program `leftshift.asm` that performs the left shift operation on the word in RAM[0] a total of RAM[1] times, and stores the result in RAM[2], i.e. RAM[2] $\leftarrow$ RAM[0] $\ll$ RAM[1]. You may assume RAM[1] is non-negative. Can you see how to optimise when RAM[1] ≥ 16?

Codestuff.online link: Here!

In a *left rotation* operation, like a left shift operation, every bit in the word is shifted one place to the left. Unlike the left shift operation, the rightmost bit becomes the old leftmost bit rather than a zero. It is as though you were reading the word off a bracelet, and you rotated the bracelet left by one bit (moving your start position one bit to the right). For example,

`1111111001111111` becomes `111111001111111`**`1`**, `1010111000011011` becomes `010111000011011`**`1`**.

Write a Hack assembly program `leftrotate.asm` that performs the left rotation operation on the word in RAM[0] a total of RAM[1] times, and stores the result in RAM[2]. You may assume RAM[1] is non-negative, and you don't need to optimise when RAM[1] ≥ 16. You may want to use your answer for the left shift exercise as a base.

Codestuff.online link: Here!

As the name suggests, the *right rotation* operation is the opposite of the left rotation operation. Every bit in the word is shifted one place to the *right*, and the *left*most bit becomes the old *right*most bit. For example,

`1111111001111111` becomes `1111111100111111`, `1010111000011011` becomes `1101011100001101`.

Write a Hack assembly program `rightrotate.asm` that performs the right rotation operation on the word in RAM[0] a total of RAM[1] times, and stores the result in RAM[2]. You may assume RAM[1] is at most 15. You may want to use your answer for the left shift exercise as a base.

Hint: There's a nice way to express right-rotation in terms of left-rotation — in fact, you can solve this problem with only a minor tweak to your left-rotation program.

Codestuff.online link: Here!

As the name suggests, the *right (logical) shift* operation is the opposite of the left shift operation. Every bit in the word is shifted one place to the *right*, and the *left*most bit becomes a zero. For example,

`1111111001111111` becomes `0111111100111111`, `1010111000011011` becomes `0101011100001101`.

We call this a right logical shift to distinguish it from a right arithmetic shift, which reads the word as a signed integer and divides it by 2. Notice that if we read the word as a positive integer, then a right logical shift and a right arithmetic shift are the same thing!

Write a Hack assembly program `rightshift.asm` that performs the right (logical) shift operation on the word in RAM[0] a total of RAM[1] times, and stores the result in RAM[2]. You may assume RAM[1] is at most 15, and you don't need to optimise when RAM[1] ≥ 16. You may want to use your answer for the left shift exercise as a base.

Hint: By far the easiest way of doing this involves starting with a right rotation! Left and right shifts are very commonly-used bitwise operations and are also relatively easy to implement in hardware, so most CPUs include them as instructions — the only reason Hack doesn't is to keep the ALU design simple.

Codestuff.online link: Here!

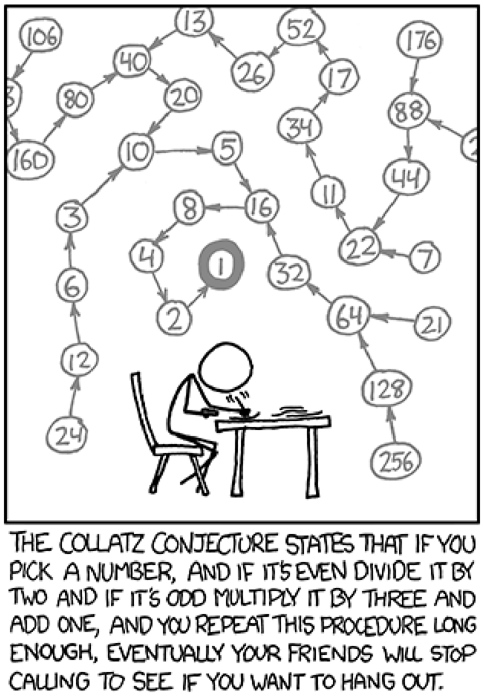
106
13
52
176
40
26
17
80
20
88
160
10
5
34
44
3
8
16
11
22
7
4
1
32
6
64
21
2
12
128
24
256
THE COLLATZ CONJECTURE STATES THAT IF YOU PICK A NUMBER, AND IF IT'S EVEN DIVIDE IT BY TWO AND IF IT'S ODD MULTIPLY IT BY THREE AND ADD ONE, AND YOU REPEAT THIS PROCEDURE LONG ENOUGH, EVENTUALLY YOUR FRIENDS WILL STOP CALLING TO SEE IF YOU WANT TO HANG OUT.

Figure 1: Source: Randall Munroe, xkcd (here). Alt text: The Strong Collatz Conjecture states that this holds for any set of obsessively-hand-applied rules.

5 The Collatz Conjecture

Consider the following iterative process. Start with a positive integer. If it's odd, multiply by three and add one. If it's even, divide it by two. Repeat this process — for example, $5 \to 16 \to 8 \to 4 \to 2 \to 1$. The Collatz Conjecture says that no matter what number you started with, you'll always eventually return to 1. Proving this is a very important open problem in mathematics.

Write a Hack assembly program `collatz.asm` to assist in verifying the conjecture experimentally. Your program should follow the process starting with RAM[0] until reaching 1, outputting each number arrived at into memory starting from RAM[32] (so that RAM[0] through RAM[31] stay free for use as variables). For example, if RAM[0] $=$ 5, then the desired output memory state is

$$\text{RAM[32]} = 5, \quad \text{RAM[33]} = 16, \quad \text{RAM[34]} = 8, \quad \text{RAM[35]} = 4, \quad \text{RAM[36]} = 2, \quad \text{RAM[37]} = 1.$$

This sort of numerical work was one of the most common applications for early computers.

Hint: For large inputs, even with how slow it is in Hack, it's much faster to do the division by 2 using a right shift than by using a naive approach. (Imagine how much faster it would be if a right shift were a single instruction!)

Modern ISAs usually have instructions for integer multiplication and division... but they also have instructions for left and right shifts, which take fewer cycles! This is worth remembering for if you need to optimise a bit of code that does a lot of multiplication or division by powers of two.

Codestuff.online link: Here!

6 Fast multiplication

A simple implementation of multiplication like the one from part 1 will multiply an m-bit number by an n-bit number in $O(2^m)$ time. Can you improve this to $O(mn)$ time? (**Hint:** Try adopting the "standard algorithm" you likely learned in school — see here. You may find bitwise operations useful!)

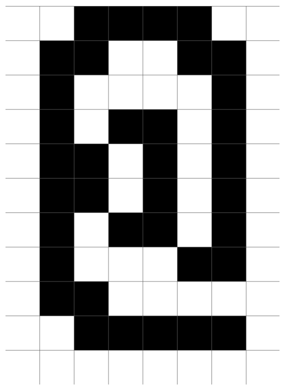
Figure 2: An example 8x11 @ sign with pixel grid.

Codestuff.online link: Still here!

7 Rogue

When dinosaurs walked the earth and your lecturers were children, computer games often used text-based graphics. One of the most famous examples is Rogue, namer of the “roguelike” genre. Rogue is played on a rectangular grid in which each cell contains a text character. The grid represents a dungeon, your character is an @ sign, each letter is a different kind of monster, and your goal is to find the Amulet of Yendor at the bottom.

Using Hack assembly, program a Rogue playfield as `rogue.asm`. Your rectangular grid should have 23 rows and 64 columns, with each cell having a width of 8 pixels and a height of 11 pixels. (There will be three rows left over; colour these black.) On startup, the @ sign should be drawn in the top-left cell of the grid. When the user hits an arrow key, the entire @ sign should move in that direction by one cell unless this would take it outside the grid.

You may find this trick useful, given that this is quite a long piece of Hack code: you can store a ROM address in a variable just like any other value, and you can later jump to that ROM address. You can use that to implement a sort of “function call” — store a label for the next bit of code in a variable `bookmark`, jump to your “function”, and then jump to the address contained in `bookmark` when your “function” has finished running to go back. This will run into issues with nested function calls, but we’ll see more about how to solve that later in the unit!