Week 5 assignment: Hack assembly practice

1 Tasks

1. Get to grips with the Hack assembler and CPU emulator.
2. Implement a multiplication algorithm.
3. Fill the screen with a toggling checkerboard pattern.
4. Implement a right shift algorithm and use it to investigate the Collatz conjecture.
5. 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 official documentation is available from the unit page. 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 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.

3 Getting started

Load the pre-existing assembly program add.asm from lectures (available from the unit page). Turn it into a .hack file using the assembler, and then run it using the CPU emulator. Load the accompanying test script add.test and verify that it runs.

Write a Hack assembly program that subtracts RAM[1] from RAM[0] and stores the result in RAM[2]. You may assume there is no integer overflow. If you're having trouble with this, take another look at add.asm and video 5-2.

4 Multiplication

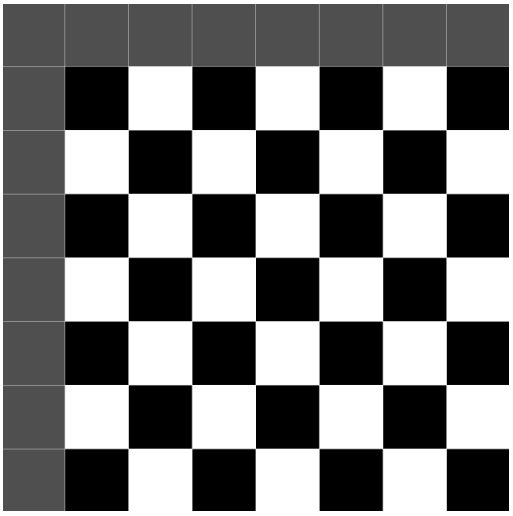
Write a Hack assembly program that multiplies RAM[0] by RAM[1] and stores the result in RAM[2]. You may assume there is no integer overflow and that both RAM[0] and RAM[1] are non-negative. (You may want to start by writing pseudocode and/or C code to solve the problem, then translate it into assembly.)

Note: In debugging your code, you will probably find it helpful to set breakpoints — you can do this via the red flag icon. You can break at a given line of assembly by setting a breakpoint for the PC with the appropriate value. You will also probably find that the default CPU simulator speed is too slow. Remember that you can run the CPU simulator extremely fast by changing "Animate" from "Program flow" to "None" in the top bar. This will stop you from editing memory manually until you change it back, although you can still clear memory using the blank paper icon (next to the binoculars).

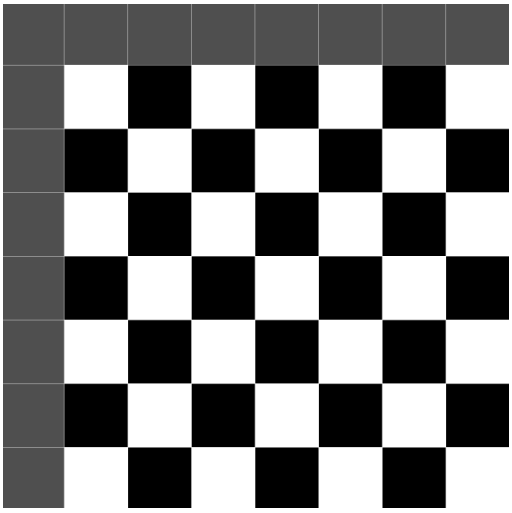
Extra credit: A simple implementation of multiplication will multiply an m-bit number by an n-bit number in $O(2^m)$ time. Can you improve this to $O(mn)$ time? (**Hint:** Try adopting the "standard algorithm" you likely learned in school — see here. You may find bitwise operations useful, so I'd recommend trying this after finishing that section of the sheet.)

5 A prettier screen fill

Write a program that behaves as follows. While no key is pressed, the screen should be filled with a checkerboard pattern as shown in Figure 1. The top-left pixel of the screen should always be black. While the "c" key is held, the



(a) The checkerboard pattern while "c" is not held.



(b) The checkerboard pattern while "c" is held.

Figure 1: Left: Examples of the checkerboard patterns for Section 5.

screen should be filled with the opposite checkerboard pattern, with the top-left pixel of the screen coloured white. The top-left pixel should be coloured correctly no matter when "c" is pressed.

In testing, you will almost certainly find that the default CPU simulator speed is too slow. Remember that you can run the CPU simulator extremely fast by changing "Animate" from "Program flow" to "None" in the top bar. This will stop you from editing memory manually until you change it back, although you can still clear memory using the blank paper icon (next to the binoculars).

Extra credit: Modify your program to toggle the checkerboard pattern being displayed every time the "c" key is pressed, rather than requiring it to be held. You will need to make sure your program only registers each separate press of "c" once to avoid uncontrollable flashing.

6 Fun with bitwise operations

6.1 Left shift

Consider a binary word. In a *left shift* operation, written $\ll$ in C, every bit in the word 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`.

Write a Hack assembly program 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, and you don't need to optimise when RAM[1] ≥ 16.

Hint: While the Hack CPU doesn't support left shifts natively, they can be expressed nicely in terms of an operation it does support. The part of your code that actually performs the left shift should be very short. You should try and work this out for yourself, but if you are having trouble, here is the solution in ROT13 (decoder here): Gb fuvsg k yrsg ol bar cbfvgvba, nqq k gb vgfrys.

6.2 Left rotation

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`		`1010111000011011`
becomes	`1111110011111111`,	becomes	`0101110000110111`.

Write a Hack assembly program 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.

Hint: The difference between a left shift and a left rotation depends only on the most significant bit of RAM[1]. The Hack CPU already provides an easy way of testing whether that bit is 1 — can you see what it is?

6.3 Right rotation

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

Write a Hack assembly program 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.

6.4 Right (logical) shift

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`		`1010111000011011`
becomes	`0111111100111111`,	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 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.

6.5 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 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

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

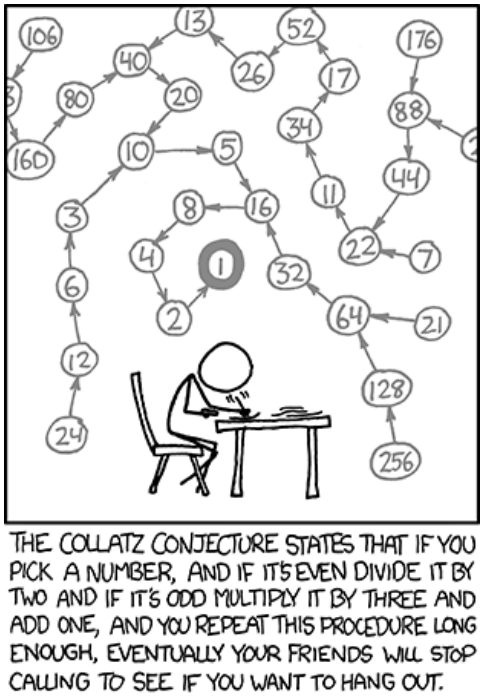


Figure 2: Source: Randall Munroe, xkcd (here). Alt text: The Strong Collatz Conjecture states that this holds for any set of obsessively-hand-applied rules.

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.

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. 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!

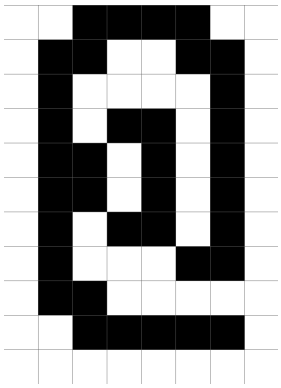


Figure 3: An example 8x11 @ sign with pixel grid.