Week 5 assignment: Hack assembly practice (part 1)

1 Why are there two parts?

The first part of this assignment, on this sheet, covers what you need to know for the exam. The second part is a collection of harder questions for practice and revision. **You should try very hard to have the first part complete before the start of week 8** as from that point onwards everything in the unit will be built on your understanding of assembly. Don't worry if you don't finish the second part, though, we intentionally put more on there than would be reasonable for a week's work!

2 Tasks

1. Write some basic Hack assembly programs.
2. Learn to test and debug code using the Hack assembler and CPU emulator.
3. Learn about bit masking.
4. Learn to use the screen and keyboard in Hack assembly.

3 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, although we can't guarantee full support for Mac OS in this unit because we don't have a good test environment.

In labs, you can run this software by opening a terminal and entering "`module load nand2tetris`" followed by "`Assembler.sh &`" and "`CPUEmulator.sh &`".

In order to run this software (and anything else from the nand2tetris suite) on your home computer, you will need to download and install it yourself. As with Logisim, you will need to install the Java Runtime Environment if you haven't already — see here. 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.

4 Getting started

You'll start with some simple tasks to get used to the basic functionality of the assembler and CPU emulator.

- Load the pre-existing assembly program `add.asm` from lectures (available from the unit page). Turn it into a file `add.hack` using the assembler.
- Load `add.hack` in the CPU emulator. In the emulator, write 3 into RAM[0] and 4 into RAM[1] and run the code using the "run" button (the double arrow). Hopefully, it will store 24 in RAM[2].
- Notice how slowly the emulator is running. Try adjusting the animation speed slider in the top bar to make it faster — you can clear memory for a second run using the button just above the RAM display. (Don't be the guy from a previous year who complained after test 2 that the practical part was totally unreasonable because the emulator was far too slow...!)
- You should also try setting the "Animate:" dropdown to "No animation" for no delay at all. This can be useful for larger files, **but it will prevent you from editing values in RAM, and the RAM values the emulator displays will not update while the program is running.** This is a common gotcha — don't let it catch you out in the exam.

- Rather than using the assembler, try loading add.asm into the CPU emulator directly. It works just fine, it's a lot faster in an exam, and in fact this is often the better way to do things as explained in the next section. But understanding viscerally that the assembler can translate a .asm file into machine code which can be loaded into ROM directly is important — when we talk about how the hardware works in week 7, it will be reading machine code, not assembly, and in week 8 we will talk about how the assembler itself works. The assembler can also be useful for debugging, again as we'll see in the next section.

Now try writing some code of your own. Write a Hack assembly program `sub.asm` 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. Test it in the CPU emulator.

In these labs, there's another way of testing code as well that runs it through multiple test cases — go to codestuff.online here and paste it in. It'll give you a good indication whether your solution is correct, but once you've found a test case that fails, it's still easiest to debug that test case using the CPU emulator.

5 Multiplication and debugging

Now you'll try writing a slightly more complex program. It will probably fail the first time you run it, because programming is hard. **Do not try to debug it yet**, but instead move on to the next part of the section to learn what tools you have available. At the end we'll circle back to get it working.

Your goal is to write a Hack assembly program `mult.asm` 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. **Before you start writing**, work out how you would do this in C without using the $\star$ operator. Then translate that C code into assembly — video 5.3 is a useful reference here for how to translate loops and conditionals into Hack. Everyone writes code differently, and you don't have to keep working this way after this exercise, but in John's experience it's very helpful. Doing things this way lets you think about "what do I need to do?" separately from "how do I express that in Hack?", and he finds it lets him work significantly faster as a result unless what he's trying to do is very easy relative to his experience with Hack. The pseudocode also gives you a useful set of comments to navigate through your code when you start debugging, and it's helpful to start thinking about the process of "compiling" code from one language to another early in the unit.

Before you test your multiplication code, download `odd.asm` from the unit page. This code contains some intentionally planted bugs — you'll fix these to learn about the available debugging tools. `odd.asm` is intended to behave as follows. RAM[0] will be a positive number. Store RAM[0] in RAM[1] if RAM[0] is odd, and otherwise store 42.

- First try loading `odd.asm` into the CPU emulator directly. You should get an unhelpful error message. Whenever the CPU emulator refuses to load an asm file, this is because the assembly is malformed in some way — there's a syntax error. To find out where, fire up the assembler and click the "Fast translation" button (second from the right at the top bar) — this is much faster than stepping through line-by-line. This should highlight the line with the problem. Fix it, then repeat the process until the asm file successfully compiles into a hack file.
- At this point, you no longer need the assembler and can work with the CPU emulator directly. Open `odd.asm` in the CPU emulator, and go to the dropdown just above the ROM display and toggle it between "Asm" and "Sym". In "Asm" mode, the ROM display shows the assembly code as executed by the CPU from a hack file, with all variables replaced by RAM addresses, all labels replaced by RAM addresses, and all comments removed. In "Sym" (symbolic) mode, the ROM display shows the assembly code as written in the original file, with variable names intact and label names displayed by their corresponding ROM address. End-of-line comments are also shown.
- Using "Asm" and "Sym" mode, find the bug in the test case $\mathrm{RAM}[0] = 501$, which should store 42 in RAM[1]. You'll find the "single step" and "step back" buttons on the top bar helpful for this — note that the "step back" button is only active outside "No animation" mode.

  While this particular bug is a little mean and artificial, this *type* of bug is a very common issue in Hack assembly code and you should know what the symptoms are. Find it, fix it, and get this test case working.

- Now look at the test case of $\mathrm{RAM}[0] = 500$, which should output 500. First run the test case with no animation to get a quick answer, and notice that it's wrong. But waiting for the whole loop to run is painfully time-consuming, and nothing seems to go wrong in the first couple of iterations. The solution is to set a breakpoint, a concept you've probably seen (or will see) in Overview of Software Tools.
  - Switch the CPU emulator to symbolic mode to orient yourself in the code, and scroll down in ROM to the end of the loop at ROM[15]. Right-click this word of memory — it should turn red.
  - Now run the program in "No animation" mode. It should run to ROM[15] and then stop. Set the animation mode back to "Program flow" so you can step backwards if you want to, then step through looking for the point where things break.
  - Find the bug and fix it.

  Any decent IDE will let you set conditional breakpoints in your language of choice. They are a hugely useful tool. You can also set conditional breakpoints by clicking the flag icon in the top bar — useful if something is going wrong in iteration 357 of a 500-iteration loop.

Now go back to your multiplication code and test it on codestuff.online here. If it breaks, excellent! Use what you've just learned to fix the bugs. If not, don't worry, I'm sure you'll have an opportunity to try debugging for real soon enough.

(Also, if you didn't leave comments in your multiplication code, you're probably regretting that fact now that you have to fix a bug after some time away from it. Take this as a learning experience!)

6 Masking

Often we want to affect a binary word *as* binary, rather than as the number it contains — we want to flip some of its bits, or set some of its bits to 1 or 0, or shift the whole thing a few places to the left or right. For this we use *bitwise operations*. These work just like multi-input gates from the first part of the unit — if $x = x_1 \dots x_{16}$ and $y = y_1 \dots y_{16}$ are 16-bit words, then e.g. the i'th bit of $x \wedge y$ is $x_i \wedge y_i$. Any reasonable modern programming language will give you access to bitwise NOT, AND, OR, and XOR operations; in C the operators for these are `~`, `&`, `|` and `^`, respectively. Hack assembly gives you access to bitwise NOT, AND and OR.

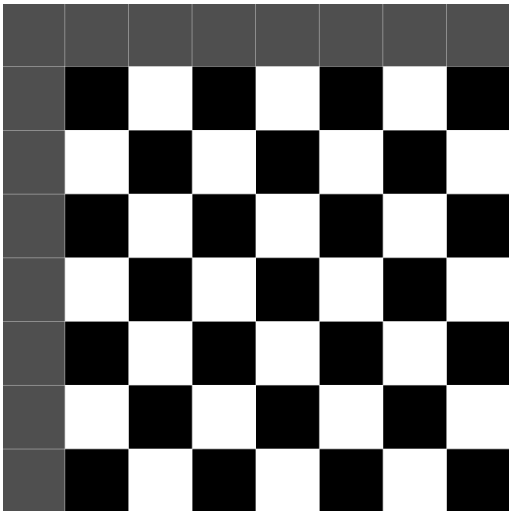
We use AND operations to set bits low, using the fact that $x_i \wedge 0 = 0$ for any value of x_i. To try this out, write an assembly program `and.asm` that counts the number of bits of RAM[0] that are set to 1, then writes the result to RAM[1] — for example, if $\mathrm{RAM}[0] = 11 = 0b1011$, then RAM[1] should be set to 3. When testing your code in the CPU emulator, you may find it useful to set the "Format" to binary in the top-right of the window to see the individual bits of words in memory.

Codestuff.online link: Here!

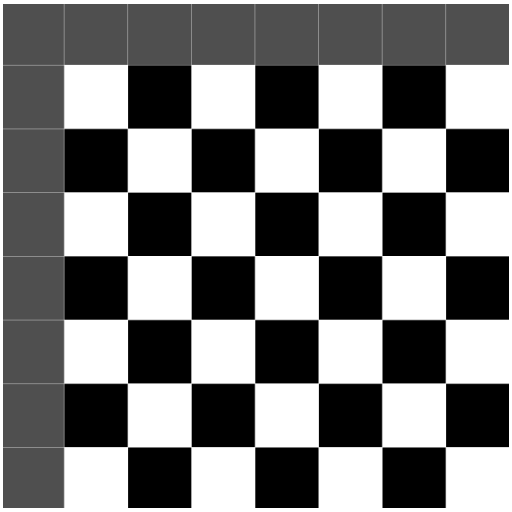
In much the same way, we use OR operations to set bits high, using the fact that $x_i \vee 1 = 1$ for any value of x_i, and XOR operations to toggle bits, using the fact that $x_i \oplus 1 = 1 - x_i$ for any value of x_i. Hack doesn't give you a direct XOR command, but of course this shouldn't stop you from using XORs — you can always build them out of ANDs, ORs and NOTs. To try this out, write an assembly program `xor.asm` that flips the third and fifth-most significant bits of RAM[1] if RAM[0] is divisible by 256 (i.e. if all of the eight least significant bits of RAM[0] are zero). For example, given an input of $\mathrm{RAM}[0] = 512 = 0\text{x}0200$, if $\mathrm{RAM}[1] = 2831 = 0\text{x}0\text{B}0\text{F}$ initially, then RAM[1] should be changed to $8975 = 0\text{x}230\text{F}$.

Codestuff.online link: Here!

Whenever we AND, OR or XOR an unknown word with a fixed word to change the values of certain bits, we call the fixed word a *mask*. When a bit is used to store a boolean variable, we call it a `flag`; setting that bit to 1 is called *setting* the flag, and setting it to zero is called *clearing* the flag.



(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 7.

7 Input and output in Hack

You should definitely review the video on writing to the screen and reading from the keyboard before you try this section!

First, as a proof of concept, write a program `checker1.asm` that behaves as follows. While no key is pressed, the top-left pixel of the screen should be coloured black and all other pixels should be coloured white. While the "c" key is pressed and held, the bottom-right pixel of the screen should be coloured black and all other pixels should be coloured white. Some points to be careful of:

- As explained in lectures, `@x` commands are only valid when `x` is a variable, a label, or a **15**-bit positive integer — so e.g. `@65535` is not valid Hack even though $65535 = 0\text{xFFFF}$ fits in a word of Hack memory. When outputting to the screen, you'll often need to set memory to 16-bit integers, so you should use either masking or negation (`M=-M` or similar).
- If your first attempt puts the pixels in the wrong position on the screen, **stop, go back over the video, and think**. Don't just start trying stuff blindly.
- To test your program in the emulator, you should run it in "no animation" mode. For keyboard input, click the keyboard button below the screen output while the program is running. You will probably find conditional breakpoints useful for debugging!

Codestuff.online link: Here!

Now write a program `checker2.asm` 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 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.

Before you start writing your code, you should work out exactly what values you need to write to which RAM addresses to display each pattern — check the video again if you need to. Test you've got that right by making a much simpler tiny Hack program that writes those values manually to a few cells. This will save you a lot of pain if your understanding of how screen output works is wrong.

Codestuff.online link: Here!

Since reading from the screen and writing to the keyboard requires you to understand memory indirection, conditionals,

loops and binary representations, it's a good overall test of your assembly skills, and there's normally a 15-mark question in the exam asking you to do something like this.