Problem 1: HMMM Assembly

courses.edx.org/courses/course-v1:HarveyMuddX+CS005x+2T2016/courseware/76469178cd4f467c9527a3fe617e3762/96ada6e8723a4019b0b77a3b41ef9580/

This problem is all about programming in assembly language on a small computer model called HMMM, the "Harvey Mudd Miniature Machine".

There are two parts:

- First, we give you a bit of background. This short preamble will help you avoid a lot of potential pitfalls.
- Next, you're asked to write a HMMM program that counts down to zero, starting from the *cube* of a positive input.

Here is the HMMM documentation, which includes all of the instructions in the HMMM language.

Happy HMMMing!

When you're finished, submit your code at the bottom of this page.

Introduction to Assembly Language

Although the HMMM model is not a physical computer, its design is fundamentally very similar to that of modern real computers.

Why not use a "real" computer? It would simply take too much time to learn the details of a real processor's instruction set. What's more, those details won't apply to another computer's instruction set. The instructions in HMMM are common to **all processors**.

HMMM has about 20 instructions. Modern computers typically have between twice and twenty times as many instructions.

What's the point of programming in assembly language??

When a computer is first built, all it can do is execute machine-language instructions written in binary. Programming in this binary machine language is a pain! Therefore, the first thing that the designers of a computer (such as the HMMM) typically do is write an "assembler".

With the assembler, programmers can now write programs with instructions such as `add`, `jump`, and the others in HMMM. The assembler converts these instructions into their corresponding binary equivalents, aka the "machine language" that the computer's circuits execute.

Once an assembler exists, the next step is to use assembly language to write more powerful languages such as Python.

Don't worry—we won't ask you to implement Python in HMMM! But writing a few short assembly-language programs will give you a sense of how one would start building such tools in the processor's native language.

There's another reason for understanding assembly language. Whenever you run a program in Python (or Java or C or any other programming language), that program is ultimately compiled into assembly/machine language so that the computer can run it. Learning to program in assembly language will allow you to understand what's **really** going on inside the machine when your program is run.

What's the difference between registers and memory?

HMMM has 16 registers named `r0` through `r15`, and 256 RAM locations ("memory"). A real processor might have have (about) the same number of registers as HMMM—maybe a couple of times bigger—but would have "lots and lots" of RAM memory—typically on the order of millions or billions of memory locations that it could access.

Registers are where the computer actually ***does*** its computation. Registers are the limited amount of data the computer can keep in its "head."

RAM Memory is where all of the other data—***and the program itself***—are located. Think of RAM as a "notebook" where the computer keeps lots of information that it can't otherwise remember.

Keep in mind that the actual program is located in memory and that the computer fetches one instruction at a time from that RAM memory into its "brain" (registers) so that it can actually execute that instruction. This process of fetching instructions from memory to execute on the processor is known as the *Von Neumann* architecture. All modern computers use the Von Neumann architecture.

A closer look at HMMM instructions

Each instruction typically tells the computer to do something with its registers (add numbers, etc). When that instruction is done, the computer fetches the next instruction from memory. By convention, the program resides in memory beginning at memory location 0.

As in most computers, most HMMM instructions "operate" on *registers*. If you look over your class notes, you'll see there are instructions such as `add` that take three registers as arguments, e.g.,

```
add r3 r1
r2
```

The values in the right two registers, `r1` and `r2`, are added together and the result is stored in the left-side register `r3`.

This instruction is Python's `r3 = r1 + r2`. Similarly, the `sub` (subtract), `mul` (multiply), and many other instructions operate entirely on registers.

There are a few exceptions, however! The most important are `jumpn`, `addn`, and `setn`. Note that each of these instructions ends in the letter n. This means that each takes a *raw number* as an argument. For instance, the number supplied to the `jumpn` instruction indicates the line to which the program will jump next. Jumpn's relatives `jltzn`, `jgtzn`, `jnezn`, and `jeqzn` accept a register *and* a raw number. For example, `jltzn r2 42` will jump to line 42 *only if the value in register* `r2` *is less than zero*.

Feeding raw numbers into the instructions `addn` and `loadn` works in a very similar manner. Simply put, the command `addn r5 42` adds the number `42` to the pre-existing contents of register `r5` and then stores the result back in register `r5`.

Note that the instruction `addn rX -1` is an easy way to subtract 1 from a register!

The instruction `setn r3 42` simply loads the number `42` into register `r3`—this is also useful….

Let's get started with HMMM!

Running HMMM's Assembler and Simulator

We're going to use an excellent online implementation of HMMM created by Sean Hickey, a computer science teacher in Minneapolis, Minnesota. The easiest way to work on this assignment is to open the online HMMM editor linked above in a new tab, so you can switch between your HMMM program and these instructions easily.

Try It!

First, copying and pasting the following code into the HMMM editor:

```
# counting up
00 read r1          # get # from user to r1
01 write r1         # print the value of r1
02 addn r1 1        # add 1 to r1
03 jumpn 01         # jump to line 01
04 halt             # never halts! [use ctrl-
c]
```

Then, press the blue "Assemble!" button. You'll see lots of 0s and 1s appear in the "Binary" section on the right.

Now, press the "Simulate!" button on the right. This will take you to another page, where you can see the output, CPU, and RAM.

If you need to edit your HMMM code, you can press the "Back to Editor" button on the left. If you do edit your HMMM, you'll need to assemble it before you try to simulate it.

In this new page, press the "Run!" button to run your HMMM program. The program will start executing. It will ask you to enter a number in a dialog box (this is the effect of the `read r1` instruction at the start of the program). Type in `42` or your favorite number and hit return. You should see the simulator counting upward from `42` or whatever number you typed in.

Look over the code to be sure you understand how it works!

Another Example

Now, return to the HMMM editor by clicking the "Back to Editor" button at the top left of the page, above the output. This will take you back to the editor, where you can make changes to your code.

For now, paste in this new HMMM program:

```
# multiplication example
00 read r1          # get # from user to
r1
01 read r2          # get # from user to
r2
02 mul r3 r1 r2     # r3 = r1 * r2
03 write r3         # print what's in r3
04 halt             # stop
```

Make sure to press the "Assemble!" button before returning to the HMMM Simulator!

Read this program instruction-by-instruction so that you understand what's happening in it. This program shows how to take in multiple inputs and use them in a subsequent computation!

The Cubic Countdown

For Homework 7, Problem 1, your task is to change the multiplication example code from the second example above so that it does the following:

1. First, it should ask the user for an input. For this problem, you may assume that the input will be ***non-negative*** and less than 30.
2. Next, your HMMM program should compute the ***cube*** of that input. It should print the result. You'll need *multiple* multiplications to compute the cube!
3. Next, it should count ***downward*** from that resulting value (the cube of the input) one integer at a time. It should print each one until it gets less than zero.
4. When this countdown is less than zero, the program should stop. The last value printed should be `0`, not `-1`.

Hints

A good way to do this problem is to write ONE step in the above list at a time and TEST your program to be sure that step works each time. For example, you might:

- First, write a program that takes in an input, computes and prints its cube, and then exits.
- Then, write a program that takes in an input, computes and prints its cube, and then prints *one less than that cube*, then exits.
- Finally, you might consider how to add a loop that will continue this process all the way to zero before exiting.

You don't have to use this development process, but the one-step-at-a-time approach is particularly crucial for assembly language, because it's difficult (at least for us humans!) to keep track of where everything is.

Commenting

Commenting is a must!

For this same reason—that keeping track of what is going on each step of the way is quite difficult for human readers—all of your HMMM code should have a comment on ***every line*** except for null-operations, `nop`s, which are convenient to use as spacers so that you have room to grow your program without renumbering lines in the future.

Submit Homework 7, Problem 1

10.0/10.0 points (graded)

To submit your Homework 7, Problem 1 code, you'll need to copy it from the HMMM editor and paste it into the box below. After you've pasted your code below, click the "Check" button.

IMPORTANT: Make sure that there aren't spaces at the beginning of your code, and that you copied all of the characters. If there are extra spaces or you are missing spaces, our server won't be able to run your code and we won't be able to give you any of the points you deserve for your hard work.

1

```
# THE CUBIC
COUNTDOWN
```

2

```
00 read r1          # get # from user to
r1
```

3

```
01 mul r2 r1 r1     # r2 = r1 *
r1
```

4

```
02 mul r1 r2 r1     # r1 = r2 *
r1
```

5

```
03 write r1         # print what's in
r1
```

6

```
04 addn r1 -1       # add -1 to
r1
```

7

```
05 jgtzn r1 03      # if r1 > 0 jump to line
03
```

8

```
06 write r1          # print what's in
r1
```

9

```
07 halt              #
stop
```

Press ESC then TAB or click outside of the code editor to exit
correct

correct

Test results
CORRECT See full outputSee full output

You have used 1 of 3 attempts Some problems have options such as save, reset, hints, or show answer. These options follow the Submit button.