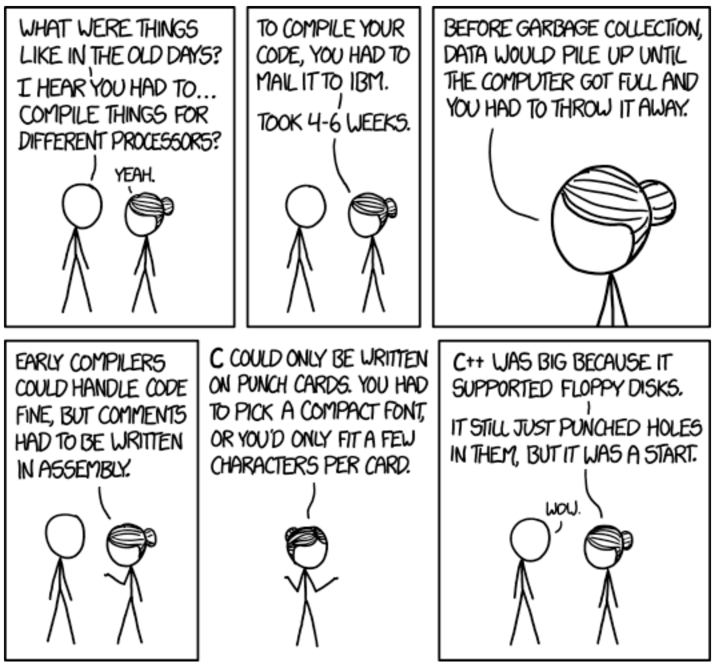
CS51 - Assignment 5

Due: March 5th, at 11:59pm


WHAT WERE THINGS LIKE IN THE OLD DAYS? I HEAR YOU HAD TO... COMPILE THINGS FOR DIFFERENT PROCESSORS?
YEAH.
TO COMPILE YOUR CODE, YOU HAD TO MAIL IT TO IBM.
TOOK 4-6 WEEKS.
BEFORE GARBAGE COLLECTION, DATA WOULD PILE UP UNTIL THE COMPUTER GOT FULL AND YOU HAD TO THROW IT AWAY.
EARLY COMPILERS COULD HANDLE CODE FINE, BUT COMMENTS HAD TO BE WRITTEN IN ASSEMBLY.
C COULD ONLY BE WRITTEN ON PUNCH CARDS. YOU HAD TO PICK A COMPACT FONT, OR YOU'D ONLY FIT A FEW CHARACTERS PER CARD.
C++ WAS BIG BECAUSE IT SUPPORTED FLOPPY DISKS.
IT STILL JUST PUNCHED HOLES IN THEM, BUT IT WAS A START.
WOW.

`https://xkcd.com/1755/`

Overview

- For this assignment, we will be coding up functions in assembly for the CS51 machine.
- Each problem will be in its own file, so you will be creating four files: `negate.a51`, `rectangle_perimeter.a51`, `multiply.a51` and `count_ones.a51`.
- Like previous assignments, there is a pre-lab portion that needs to be completed before lab on Friday.
- The end of the assignment has an appendix with a summary of the CS51 machine assembly instructions.

Running the CS51 machine simulator

Download the simulator from:
`http://www.cs.pomona.edu/classes/cs51/assignments/cs51-machine.jar`
and put it in your `cs51` folder. In theory, once you download it, you should just be able to double-click the jar and it should start the simulator. On MacOS, you likely will get the error "Apple could not verify this app is free of malware" the first time you try and run it. If this happens, do the following:

- Go to `System Settings`.
- Select `Privacy & Security`.
- Scroll down to the bottom of this window (on the left) to the "Security" section.
- You should see `cs51-machine.jar` here with a message saying that it was blocked. Click the button "Open Anyway".

Troubleshooting

If the above doesn't work, there are a few things that you can try. Sometimes the program won't start automatically. In VS Code, in the `Terminal` window (i.e., the one where you can type commands to the OS, that usually shows a prompt ending in %), navigate to the directory where the `cs51-machine.jar` file resides (using `cd`: see Assignment 1 for a refresher). Then, to run the simulator, type:

```
% java -jar cs51-machine.jar
```

If this doesn't work and/or you get an error about Java not being installed, you'll need to install Java. To do that, go to: `https://adoptium.net/en-GB/temurin/releases`. At the top, you'll see releases for either macOS or Windows. Select the appropriate one for your OS and click the download button (circle with the arrow pointing down). Run the downloaded file to install Java.

If all else fails, come to our office hours and/or mentor sessions for help installing.

Editing and running `.a51` files

As mentioned in class, the `.a51` files are just text files with the `.a51` extension. In your `cs51` folder, create a subfolder called `assignment5` and then open this folder in VS Code. For each problem, you'll need to create a new file. Under the `File` menu, select `New Text File`. This will give you a blank text file. If you save this file, it will ask you to give it a name, and you can name it appropriately, with the `.a51` extension.

Then, add your instructions, etc. Don't forget to indent all instructions. To run your program, make sure to save it in VS Code first, then "Load" it in the CS51 machine simulator. Every time you make a change to your program, you'll need to repeat this (save in VS Code and load in the simulator).

A note on debugging in CS51 assembly

Because of its simplicity, it can be challenging to debug assembly programs. If, after looking at the code for a bit (and relevant examples from class), the program doesn't work correctly, the best way to debug is to step through the execution of your code and follow the execution. If you do this step by step, you'll eventually find the problem.

A few thoughts on this:

– Open your code in the simulator and run it one line at a time using the "step" (arrow) button.

– Get out a piece of paper and write down a place for the registers r2, r3 and the stack. As you execute each line of code, update what you think should be happening on the piece of paper, and then double-check that's what happened in the simulator. If they're different, figure out why!

– Some of the instructions that you write show up, and are executed as two instructions. In particular,

  - `psh`: is a store based on r1 (sto) followed by a decrement of r1
  - `pop`: is a load based on r1 (loa) followed by an increment of r1
  - `lcw`: loads the lower bits and then loads the higher bits
  - `jmp`: is a load (loa) followed by a function call (cal)

  Besides these four, there will be a direct one-to-one mapping between the instructions you write and the instruction view on the right. Don't get thrown off by this. You can still easily step through an instruction at a time and see your code execute.

– Remember, r1 holds the location of the next location where a value would be put on the stack. To view the stack:

  1) Look at the memory location stored in r1, i.e. the value in r1.
  2) Look at the data view on the left of the simulator and find the address (left part before the colon) corresponding to the address in r1.
  3) The address in r1 is not part of the stack. The first value in the stack is the address immediately below the address in r1 (e.g., if r1 is 00da, then the first value of the stack is 00dc). Remember, the stack grows upward toward lower memory addresses, so the stack itself will be further down, at higher memory addresses.

Using these things, you should be able to step through your code a line at a time. If you understand what's supposed to be happening, then this can be very, very helpful at identifying small bugs.

Before you come to lab

1. Download the CS51 simulator and make sure you can open it and run a program (e.g., download one of the programs from the class examples `http://www.cs.pomona.edu/classes/cs51/cs51machine/`.

2. On paper:

   Write a CS51 assembly program `positive.a51` that:

   - Prompts the user for a number
   - then prints 1 if the number is positive, 0 otherwise.

   For example, if we ran the program in the simulator and input 10, we would see the following:

   ```
   CS51 wants a value > 10
   CS51 says > 1
   ```

   Bring the paper annotated with your name to the Friday lab.

1 Warming up

Write a CS51 assembly program `negate.a51` that:

- Prompts the user for a number
- then prints the negation of the number.
- You *must* include in your program a separate function that takes as "input" (via `r3`) a number and then "returns" (again, via `r3`) the negation of that number. Your program should prompt the user for the input then call using `cal`, finally printing out the result after the function call.

For example, if we ran the program in the simulator and input 47 we would see the following:

```
CS51 wants a value > 47
CS51 says > -47
```

Reminder: For any of these programs that include a function, make sure you setup the stack. Your program should look something like:

```
    lcw r1 stack
    ...

    dat 100
stack
```

2 Rectangle fun

Write a CS51 assembly program `rectangle_perimeter.a51` that takes as input two numbers, a base and a height, and prints out the perimeter of a rectangle with that base and height.

Your program *must* include a separate function that takes as input two parameters (the base and the height) and returns the perimeter.

For example,

```
CS51 wants a value > 5
CS51 wants a value > 7
CS51 says > 24
```

3 Multiplication

Write a CS51 assembly program `multiply.a51` that takes as input two numbers and prints out the result of multiplying the two numbers together. You can assume both number are positive. We can view multiplication as repeatedly adding one of the numbers to accumulate the result. For example, in Python, we could write it as:

```
x = int(input("Enter the first number: "))
y = int(input("Enter the second number: "))

total = 0

while y > 0:
    total += x
    y -= 1

print(total)
```

In CS51, running the program would look like:

```
CS51 wants a value > 4
CS51 wants a value > 7
CS51 says > 28
```

For this program, do *not* write a separate function. As can be seen with the Python code above, we need three variables (`x`, `y` and `total`) to perform the multiplication. We can use all three available registers (`r1`, `r2`, and `r3`) to accomplish this. It's possible to do this with a function, but it requires consistent use of the stack.

4 Binary revisited

Write a CS51 assembly program `count_ones.a51` that takes as input a positive number and prints out how many 1s are in that number if the number is represented in binary.

Your program *must* include a separate function that takes as input one parameter and returns the number of 1s in that number.

For example:

```
CS51 wants a value > 13
CS51 says > 3
```

(13 is 01101 in binary)

Another example:

```
CS51 wants a value > 47
CS51 says > 5
```

(47 is 0101111 in binary)

There are a few ways that we might do this, but we're going to emulate the approach seen in the following Python code:

```
def count_ones(num):
    ones = 0

    while num > 0:
        if num % 2 == 1:
            ones += 1

        num = num >> 1

    return ones
```

We'll see if the least significant bit is a 1 and, if it is, increment our count. Then we'll right arithmetic shift one binary digit and repeat the process until the number is 0.

A few hints:

- The `sar` instruction does a right arithmetic shift in CS51 machine assembly.
- Since we're doing this as a function call, we'll only have registers `r2` and `r3` to work with. If you find that you need to use one of these registers temporarily to do some computation, you can save that value onto the stack (with `psh`), do the computation using that register, and then restore the original value (with `pop`). For example:

```
psh r3    ; save the current value of r3 on the stack
...       ; some instructions that overwrite the original value of r3
pop r3    ; restore the original value of r3
```

5 Feedback

Create a file named `feedback5.txt` that answers the questions:

- How long did you spend on this assignment?
- Any comments or feedback? What things did you find interesting, challenging, or boring?

6 When you're done

Submit your five files (`negate.a51`, `rectangle_perimeter.a51`, `multiply.a51` `count_ones.a51`), and `feedback5.txt` online using Gradescope; be sure you're uploading them to the right assignment. Note that you can resubmit the same assignment as many times as you would like up until the deadline.

7 Grading

	points
Before you come to lab	1
`negate`	1
`rectangle_perimeter`	2
`multiply`	2
`count_ones`	4
Total	10

Appendix

Below is a summary of the CS51 machine assembly instructions. Instructions can have a few types of arguments:

`R`, a register;

`S`, a signed (8 bit) number;

`B`, a label.

Here are some examples of the more common configurations:

- `RRR`: The instruction takes three registers, e.g.,

  ```
  add rx ry rz
  ```

  which translates to `rx = ry + rz`.
- `RRS`: The instruction takes two registers and a signed integer (8 bits), e.g.,

  ```
  add rx ry num
  ```

  which translates to `rx = ry + num`.
- `RR`: The instruction takes two register, e.g.,

  ```
  loa rx ry
  ```

  which translates to `rx = mem[ry]`

Below is a list of the more common instructions and their arguments:

arithmetic	`add`	`RRR` or `RRS`	addition
	`sub`	`RRR` or `RRS`	subtraction
	`neg`	`RR`	negation
bitwise	`and`	`RRR` or `RRS`	bitwise AND
	`orr`	`RRR` or `RRS`	bitwise OR
	`xor`	`RRR` or `RRS`	bitwise XOR
	`not`	`RR`	bitwise NOT
shifts	`sll`	`RRR` or `RRS`	logical left shift
	`slr`	`RRR` or `RRS`	logical right shift
	`sal`	`RRR` or `RRS`	arithmetic left shift
	`sar`	`RRR` or `RRS`	arithmetic right shift
memory	`loa`	`RR` or `RRS`	read *from* memory (also for getting a value from the user
	`sto`	`RR` or `RRS`	write *to* memory (also for printing a value to the user
	`lcw`	`RB`	load label address into register
branch	`beq`	`RRB`	branch if equal
	`bne`	`RRB`	branch if not equal
	`blt`	`RRB`	branch if less than
	`bge`	`RRB`	branch if greater than or equal
	`bgt`	`RRB`	branch if greater than
	`ble`	`RRB`	branch if less than or equal
	`brs`	`B`	unconditional branch
control	`cal`	`RR`	call a function
	`jmp`	`R`	jump
	`hlt`		halt the CS51 machine
	`dat`	`B`	make B bytes for the stack