CS 214: Introduction to computing systems
Winter Term, 2026

Lab 1

This lab will give you some practice writing assembly programs in MIPS.

Assembly using MARS

First, you should start the MARS simulator. You can download it using the link on Google Classroom (under Classwork, Resources).. MARS is both a development environment where you can write code and the simulator that lets you run it. (The simulator part is basically a program that pretends to be a MIPS processor, which we need since we don't have a real processor that runs MIPS assembly.)

On a Mac, to open the file, you might have to right-click on it in Finder and select "Open" rather than just double clicking.

Once MARS opens, you can start a new program by selecting "New" in the "File" menu or the leftmost icon in the menu at the top of the window. Then you'll be able to write a program. Begin with the following:

```
addi $v0, $zero, 10
syscall
```

This is the "simplest complete program" for MARS, which just calls the exit system call (number 10) to exit the program.

Once you've entered this program, save it (third icon from the left or using the "File" menu); save the file somewhere you can find it again (click on the "Save In" part of the file browser) and use a name ending in `.asm`. Then you need to assemble the program (analogous to compiling in Java). This is a command in the "Run" menu or the icon that looks like crossed tools. You'll get a message in the bottom panel about successful completion (or an error message, which you should address).

Once the program compiles successfully, your view will be shifted to the execute pane, which lets you watch the program being executed. (To make further changes, you first need to switch back to the edit pane by selecting "Edit" right below the row of icons near the top of the window.) You will also be allowed to run the program, either by selecting "Run" in the "File" menu or hitting the green icon with an arrow in the row of icons. When you run it, the message at the bottom of the screen will tell you that the program finished running.

Now you are ready to begin programming. I suggest that you write the sequence of increasingly-complex programs below. (Each program replacing the one before it; these basically build on each other.) Use the reference sheet to help. Note that the first couple of programs are very similar to ones that we've done in class; see if you can do them with only the reference sheet before looking at your notes or the code from class.

1. A program to print the value 5.
2. A program that reads an integer, adds 1 to it, and prints the result.
3. A program that reads an integer and prints all the integers less than its value:

```
read integer x
i=0;
while(i < x) {
  print i
  i++
}
```

You can either jam the integers together (01234...) or print spaces between them (use system call 11, which prints a char; the space character is number 32).

Remember that you can use a label to name a line by putting an identifier and colon before that line. Then the different kinds of branch instructions can jump the program to the named line. With this, the basic structure for a loop is the following:

```
       #stuff before the loop

loop:  #test condition and branch to below_loop if done
       #  (for the loop above, I'd use bge)
       #body of the loop
       b loop    #return execution to beginning of loop for next iteration

below_loop:
       #instructions to run after the loop
```

4. A program that reads an integer `x` and prints the number of values between 2 and `x-1` that divide it:

```
read integer x
count = 0
i = 2;
while(i < x) {
  if(x % i == 0)
    count++;
  i++;
}
print count
```

   This program counts the non-trivial factors of `x`. (1 and `x` itself are called trivial factors since they work for any value of `x`.) You can get the remainder by using the `rem` instruction. Again, you'll need branches to implement an if statement. (If the condition isn't true, then branch past the body of the if statement.)

5. A program that reads an integer `x` and prints `x` if it is prime. That is, if it has no non-trivial factors. (This is essentially the previous program with an if statement at the end.)

6. A program that reads an integer `y` and prints all the primes less than or equal to `y`. For example, entering `10` would result in printing `2 3 5 7`. (This is essentially a loop around the previous program.)

If you have more time, come up with other small programs that you can write using the assembly that we know.