Name: ______________________________ **Date:** ______________

CS 0447: Spring 2017 Lab 1
Due: January 20, 11:59 PM

Part 1: Getting started in MIPS assembly language

Step 1: Download the Mars simulator from http://www.cs.missouristate.edu/MARS/

Step 2: Launch the Mars simulator.

Step 3: Click on the Edit tab on the top left (in this pane, you'll see your assembly language program. Well, after you've created it).

Step 4: Go to File and click on New, to start a new program.

Step 5: Let's start with a simple program.

Start the program with:

```
.text
```

This says that the following are program instructions (and not, e.g., data). Let's do the following:

# $t4 = 3 + 6 + 4

First, let's put the value 3 in register $t4:

```
addi $t4, $zero, 3
```

The instruction above says to take what is stored in register $zero, add 3 to it, and put the result in register $t4. Register $zero ALWAYS contains 0; so this puts 0 + 3 into register $t4.

Step 6: Before you can go further, MIPS needs you to save the file.

Step 7: Now, go to the Run tab and click on Assemble.

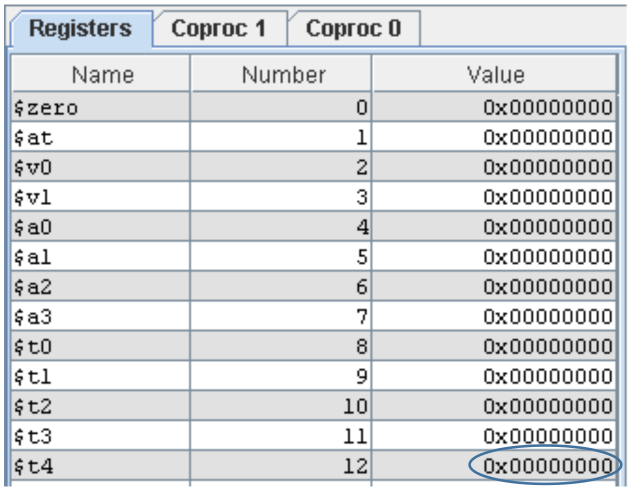
The Text Segment shows you these columns:
- **Address** -- where this instruction is stored in memory
- **Code** -- the machine code, which is 32-bits wide, of the instruction
- **Basic** -- [not too helpful at this point]
- **Source** -- the original instruction you typed

The Data Segment in the middle of the page shows you the contents of the part of memory where data is stored. We haven't put any data in memory; so the values here are all 0.

The bottom window gives messages from the simulator. Any error messages will be displayed here.

Step 8: Run the program (technically, "simulate" running the program) **one step** by pressing the F7 key or the step-forward button ▶1.

Step 9: Look at register $t4 in the panel (Figure 1) on the right-hand side of Mars's display. Mars should have highlighted $t4 for you.



Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000

Figure 1. Where to find the value of the $t4 register

Step 10: Ok, let's continue writing the program (to get back to your program, click the Edit tab).

Now we want to add the value 6 to our running sum:

```
addi $t4, $t4, 6
```

And then we want to add 4:

```
addi $t4, $t4, 4
```

Step 11: Assemble the program again. Run the entire program one step at a time by pressing F7 or . As you step through your program, verify how your program works by watching the registers and memory (if applicable).

You will see that $t4 first contains 0x00000003, then 0x00000009, and finally 0x0000000d.

The 0x just means that the number following it is in base 16 or, in other words, in hexadecimal.

Questions
Enter your answers into CourseWeb.

Question 1: What decimal (i.e., base 10) number is 0x0000000d? ___________________

Reset the simulator by pressing the F12 key or the reset button . This resets MIPS registers and memory (except the memory locations where your program instructions are stored).

Click Step under Run (or press F7 or) to step through your instructions one by one.

Question 2: Before the first instruction is executed, what is the value of the program counter register? The program counter is the register labeled "pc". Note that the program counter holds the address of the instruction to be executed.

program counter =___________

Question 3: After the first instruction is executed:

program counter = ___________

$t4 = __________

Question 4: After the second instruction is executed:

program counter = ___________

$t4 = __________

Question 5: After the third instruction is executed:

program counter = ___________

$t4 = __________

Question 6: After each instruction, the PC register is incremented by how many units? _____

Question 7: Each MIPS instruction is 32-bits wide. How wide is each MIPS instruction in these other units:

Number of nibbles or nybbles where each nybble is 4 bits = __________
Number of bytes where each byte is 8 bits = __________
Number of words where each word is s 4 bytes = __________

Question 8*: Thus, what are the units of the PC register?

a) bits
b) bytes
c) words

*To help you understand question 8 consider this: Because the PC's contents is an address of a place in memory, the PC is 'pointing' to some data there. If the PC is incremented by 1, then it is pointing to a new place in memory. What is the distance between the old and new places?

This shows a good way to learn assembly language. Enter and assemble instructions; this will show you the machine code that is produced. Then, step through the execution so you can see the effects of the individual instructions.

Part 2: Memory

Before working with memory, you need to be able to work with binary and hexadecimal numbers. Computers understand only binary numbers.

Here is an example of adding two binary numbers:

```
  0110
+ 0111
```

```
-------
  1101
```

Here is how a human does this (we'll look at hardware later in the course). Note that all numbers are in binary:

Digit 0: 0 + 1 = 1.
So, write down the 1. There is no carry.

Digit 1: 1 + 1 = 10.
So, write down the 0 and carry the 1.

Digit 2: 1 (carry) + 1 + 1 = 11.
So, write down the 1 and carry the 1.

Digit 3: 1 (carry) + 0 + 0 = 1.
So, write down the 1. There is no carry.

Question 9: Now, you try one, but with a larger number.

```
  0110011110100101
+ 0010111101011111
------------------------------
  ????????????????
```

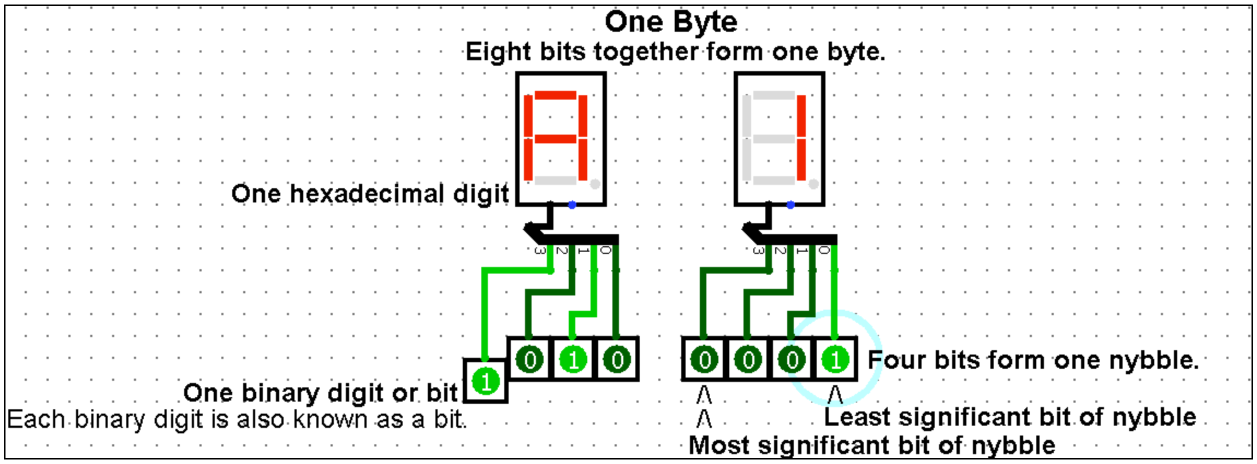

One Byte
Eight bits together form one byte.
One hexadecimal digit
0
1
0
1
0
0
0
1
Four bits form one nybble.
One binary digit or bit
Each binary digit is also known as a bit.
^
^
^
Least significant bit of nybble
Most significant bit of nybble

Figure 2. Representing One Byte

Working with long binary numbers is confusing. There must be another way! Binary numbers can be easily converted to hexadecimal by grouping the digits in groups of 4 (each group of 4 bits is called a "nybble" (Figure 2)), and converting each group individually:

Binary: 0110 0111 1010 0101
Hex: 6 7 A 5

Adding in hex is much easier. This is an example of adding two hexadecimal numbers:

```
    3A49
+   4BA9
--------
```

```
      85F2
```

Here is how a human does this. Note that numbers without the 0x are decimal:

Digit 0: 0x9 + 0x9 = 9 + 9 = 18 = 0x12 (1 * 16^1 + 2 * 16^0).
So, write down the 2 and carry the 1.

Digit 1: 1 (carry) + 0x4 + 0xA = 1 + 4 + 10 = 15 = 0xF.
So, write down the F. There is no carry.

Digit 2: 0xA + 0xB = 10 + 11 = 21 = 0x15 (1*16^1 + 5 * 16^0).
So, write down the 5 and carry the 1.

Digit 3: 1 (carry) + 0x3 + 0x4 = 1 + 3 + 4 = 8 = 0x8.
So, write down the 8.

You can check yourself using a calculator (make sure you understand why these are the right calculations!):

3 * 16^3 + 10 * 16^2 + 4 * 16^1 + 9 * 16^0 = 14921
4 * 16^3 + 11 * 16^2 + 10 * 16^1 + 9 * 16^0 = 19369
8 * 16^3 + 5 * 16^2 + F * 16^1 + 2* 16^0 = 34290
14921 + 19369 = 34290… So, above is right!

Question 10: Now, you try one. **Be sure to show your work**:

```
      3CA4
   +  1D6F
   --------
      ????
```

Now, let's look at memory. **Note that the smallest addressable unit in MIPS is one byte.** In other words, each byte in memory has its own address. Each bit (there are eight bits in a byte) shares its address with seven other bits. Start by entering the following into the simulator:

```
.data
.byte 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
```

".data" is an assembler directive (instruction to the assembler) that tells the assembler we are in the data segment of memory. ".byte" is an assembler directive that tells the assembler to store the following into subsequent bytes in memory. The data segment of memory begins at address 0x1001 0000. (Note: A space is sometimes used to separate groups of four hexadecimal digits.)

Assemble the program and look at the Data Segment window. This window shows the contents of memory as a table. To get the address of a given cell, you have to add the column label (for example, 0x1001 0000) and the row label (for example, +14) together (in this case, 0x1001 0014). Note that the + values across the top are in hex (even though the 0x is missing --- they left it off to fit more on the screen).

Also, note that each address increments from **right-to-left** in each box. In Figure 3, lines are drawn from six addresses to six specific bytes in memory (e.g., the byte having value 0x05 has address 0x10010004).

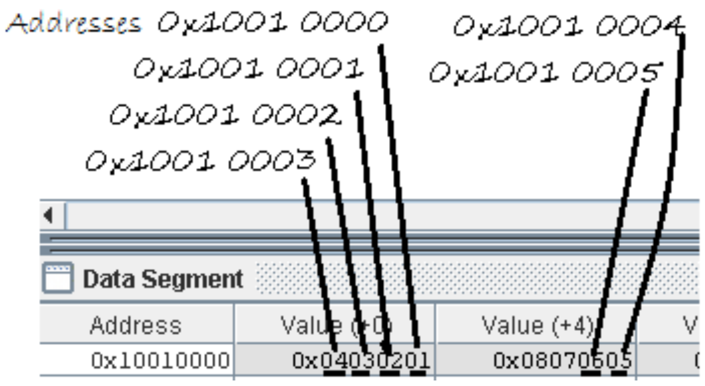


Figure 3. Example of addresses mapped to specific data items in memory

Question 11: Does each box shown in the Data Segment window represent a byte or a word? Please explain.

Note: in MIPS, each byte of memory has its own address. It's just that MARS does not show an address for every byte to fit more on the display.

Now, replace the data segment in your program with this one:

```
.data
.word 0x0A, 256, 0x15, 0x01, 32, 0xC0, 7, 0x0100, 15, 0x10, 23
```

".word" is an assembler directive that tells the assembler to store the following into subsequent words in memory. The 0x values are in hex. The other values are in decimal.

Before assembling your program, try to figure out what hexadecimal values will be stored at which memory locations by the above directives. You won't lose credit for incorrect answers here before you check the assembler --- you just need to take a stab at it. **For each word, show the correct number of hexadecimal digits! One word = 32 bits = 8 hex digits.**

Word's Value	**Address (Base 16)**	**Entire Word Shown in Data Segment (Base 16)**
0x0A	0x1001 0000	0x0000 000A
256		
0x15		
0x01		
32		
0xC0		
7		
0x100		
15		
0x10		
23		

Now assemble the code above, and look at memory.

Question 12: Now that you understand at what you are looking, **fill out the above table**. (If you have already filled out that table, then you are finished with Question 12!).