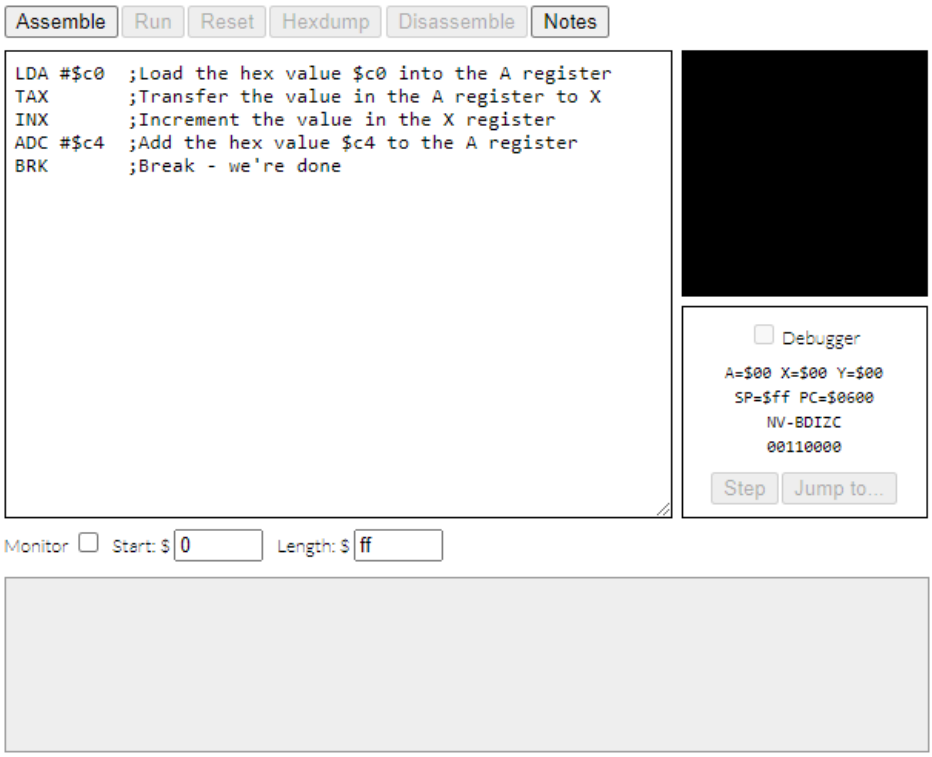
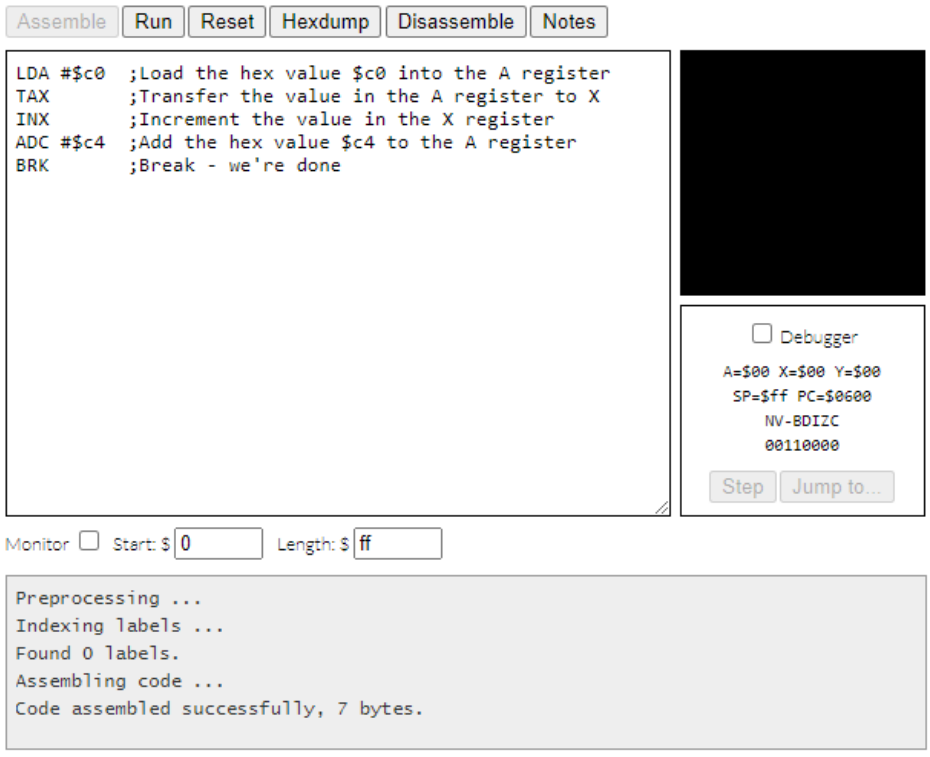
Lab Exercise #3 - 'Instructions'

Instructions in assembly language are like a small set of predefined functions. For the 6502, all instructions take either one argument or no arguments. Leading **$** symbols indicate a value specified in hexadecimal format. Here is some annotated source code to introduce a few different instructions:

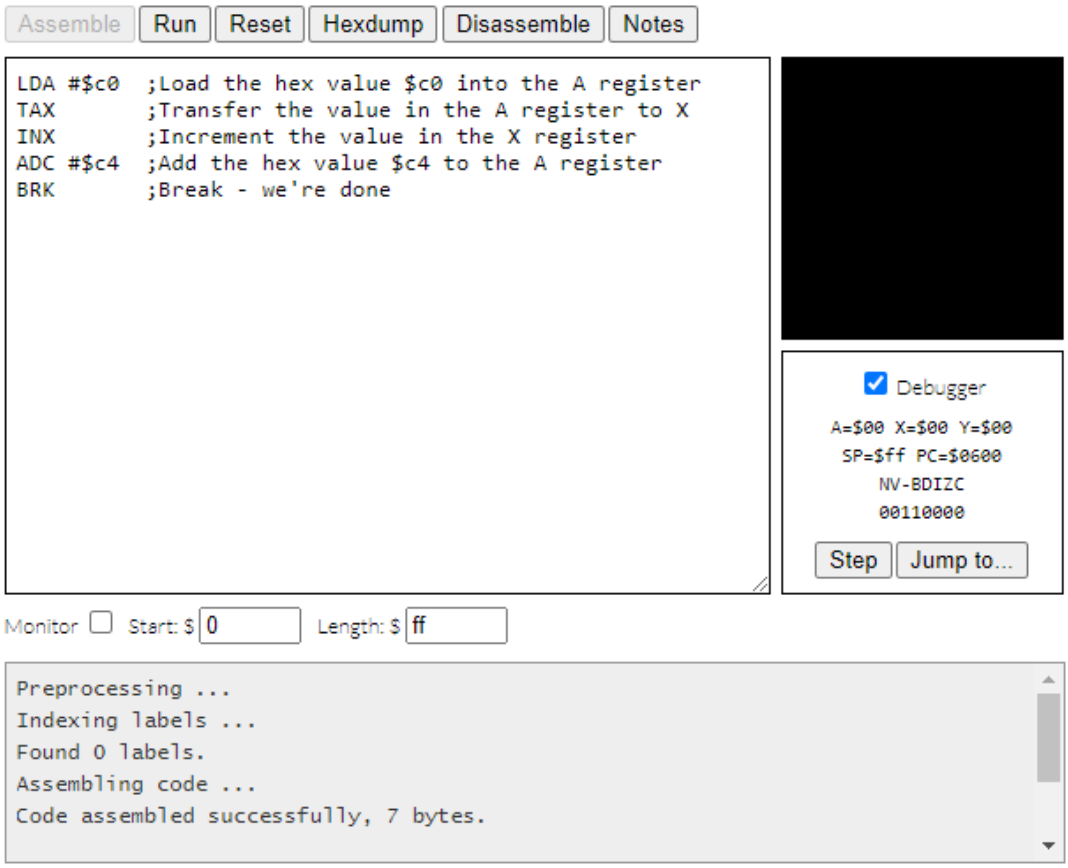
Execute the steps that follow one at a time. First, go to https://skilldrick.github.io/easy6502/ and scroll down to the section entitled “Instructions” where you should see:



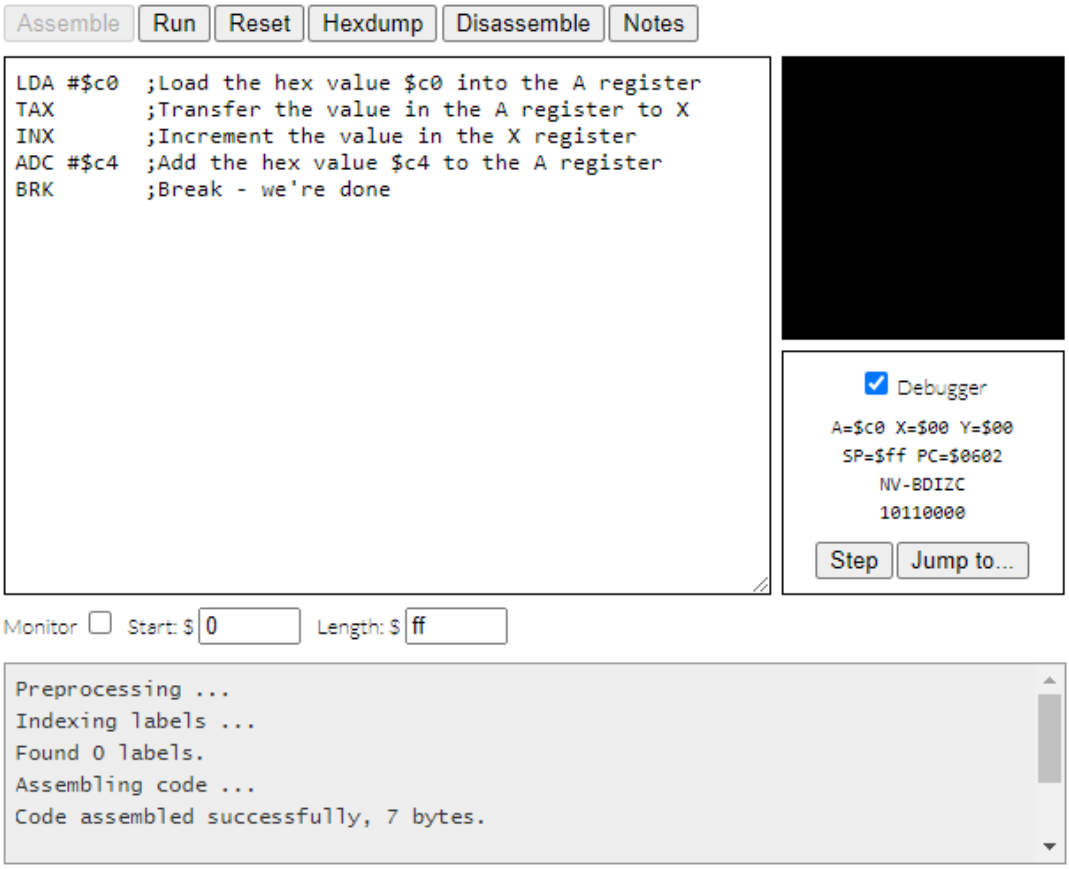
Step #1 – Click “Assemble”



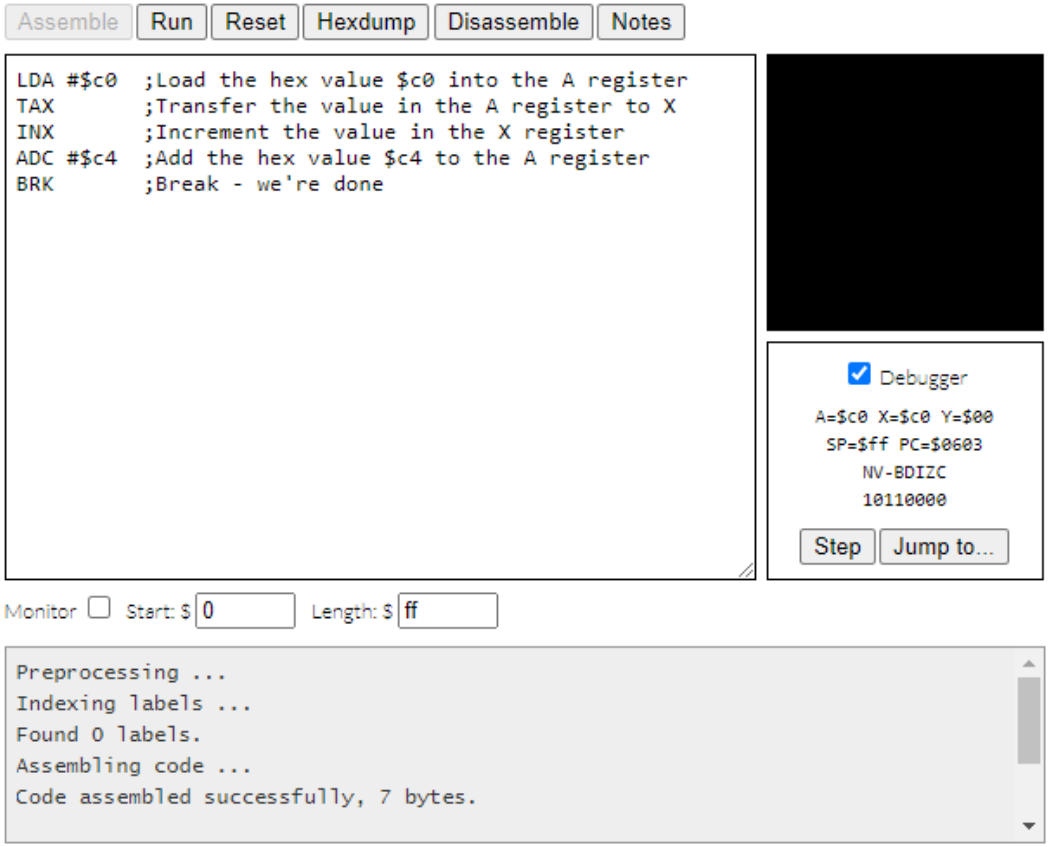
Screen above is seen after Step #1.



Step #2 Click the small box to the left of '**`Debugger`**' and see checkmark in blue outline ☑

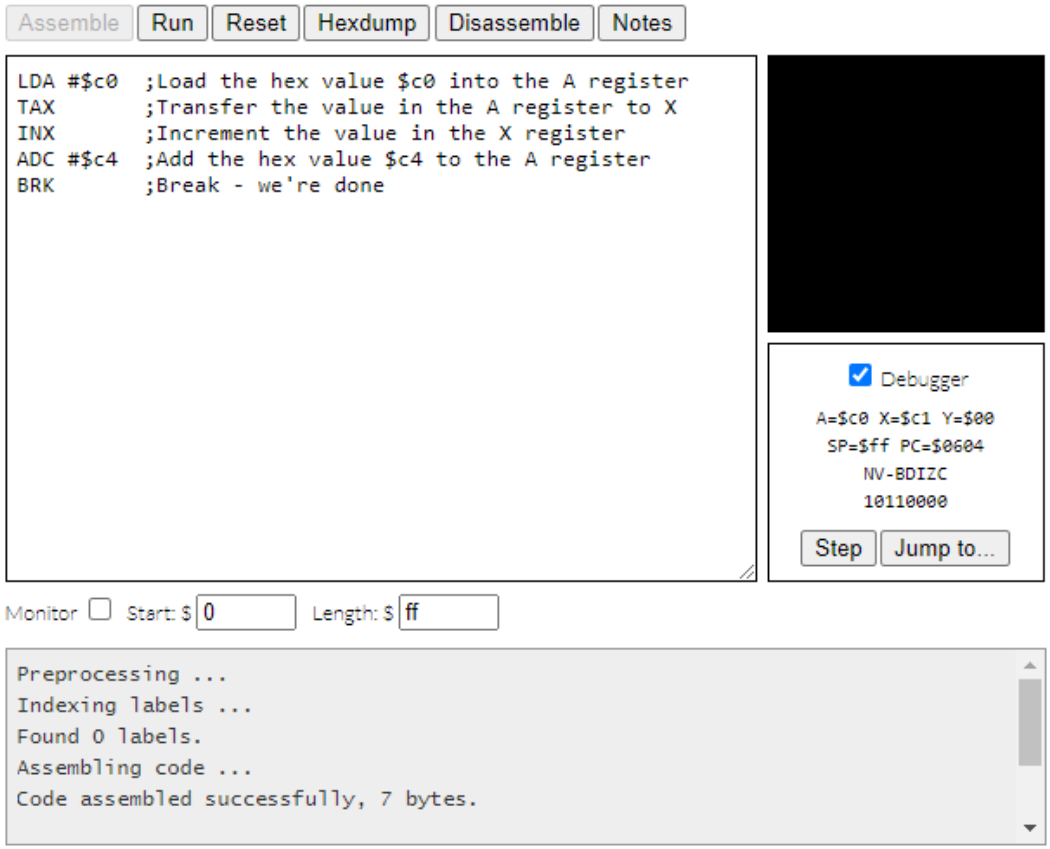


Step #3 Single-click the 'Step' button; notice the Program Counter (**`PC`**) increment to **`0602`**
Notice that the **`A`** register ("Accumulator") contains a value of **`$c0`** (decimal 192)



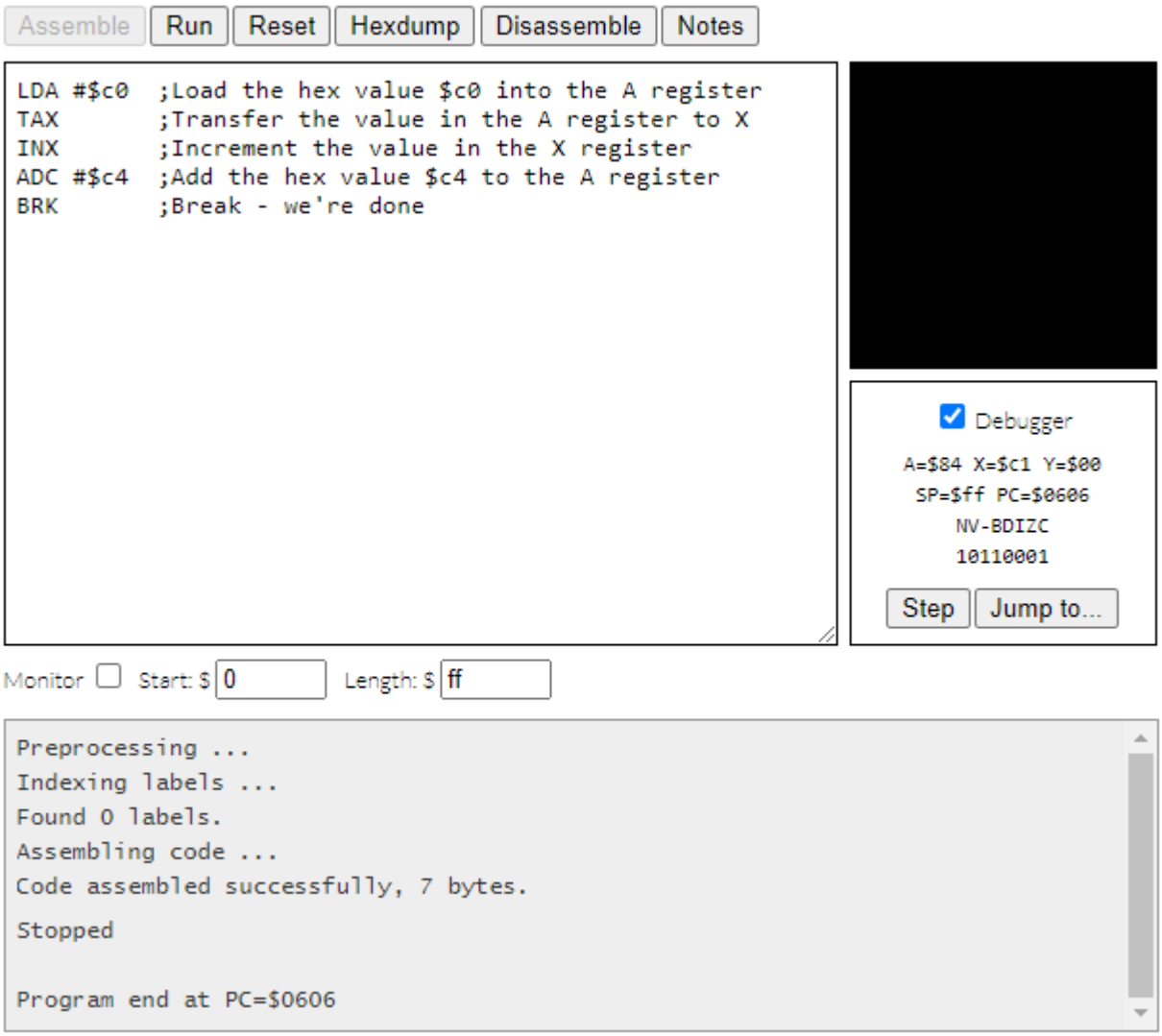
Step #4 Click 'Step' again and notice the values for register **A**, register **X**, and program counter **PC**

The value in the Accumulator (**$c0**) has been transferred to register **X**; the value of **A** is unchanged.



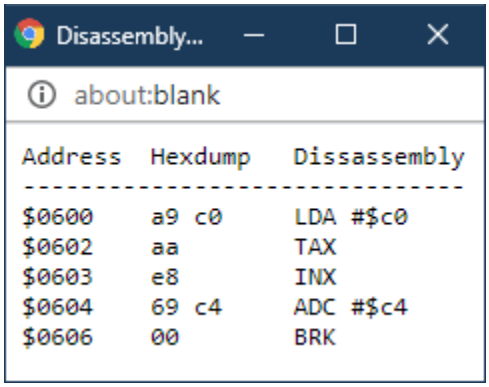
Step 5 Click 'Step' again and notice that the **X** register has been incremented to **$c1**

Notice in particular the value of the value of '**C**' (Carry) in **NV-BDIZC** Flag register: **C = 0**



Step 6 Click 'Step' again and notice that the value of the **A** register is **$84**, the low-order bits of the sum **$c0** (decimal 192) + **$c4** (decimal 196) This value should be **0x184**, (decimal 388), not **0x84** (decimal 132). What happened to the leading **1** (decimal 256) in the sum? Now look at the Flag bits. Notice that the '**C**' bit (the Carry Flag) has changed from a **0** to a **1**, indicating that we have a Carry Out of the low order eight bits. This Carry bit has a weight of **256**. Remember that the maximum [unsigned] value formed by an eight-bit-wide register is only **$ff** (decimal 255), A value greater than this (for our example, the sum is decimal 388) must be specified by using two registers.

Here is the program after disassembly. Notice the addresses for each step:



From this screenshot, you can see why the Program Counter increments as seen in the figures above

Assignment for Steps 1 through 6 above (Call it Part 1).

1. Go through each of the steps outlined above and capture a screenshot of the results of Step #6. Paste this screenshot into an 'evolving document' that you will later submit. As this document evolves, be sure to label each entry along the way since your instructor is not a psychic.
2. Review the companion document "Instruction Sequence - Instructions Example.PDF" and modify your code as directed at the bottom of the page. Assemble this code, step through the program, and capture a screenshot of its performance. Add this screenshot to your 'evolving document'.
3. Modify the original program to create a new program. For this new program, store the low-order bits of the sum in register **Y** and the **Carry** bit in register **A**. Add this screenshot to your 'evolving document'.

E-mail your assembly code to me, labeling it e.g. {Your Last Name} Lab 3 Part 1.Txt

Part 2 Writing to, and Reading from Memory

Step 7 Locate the blank editing window just below the text:

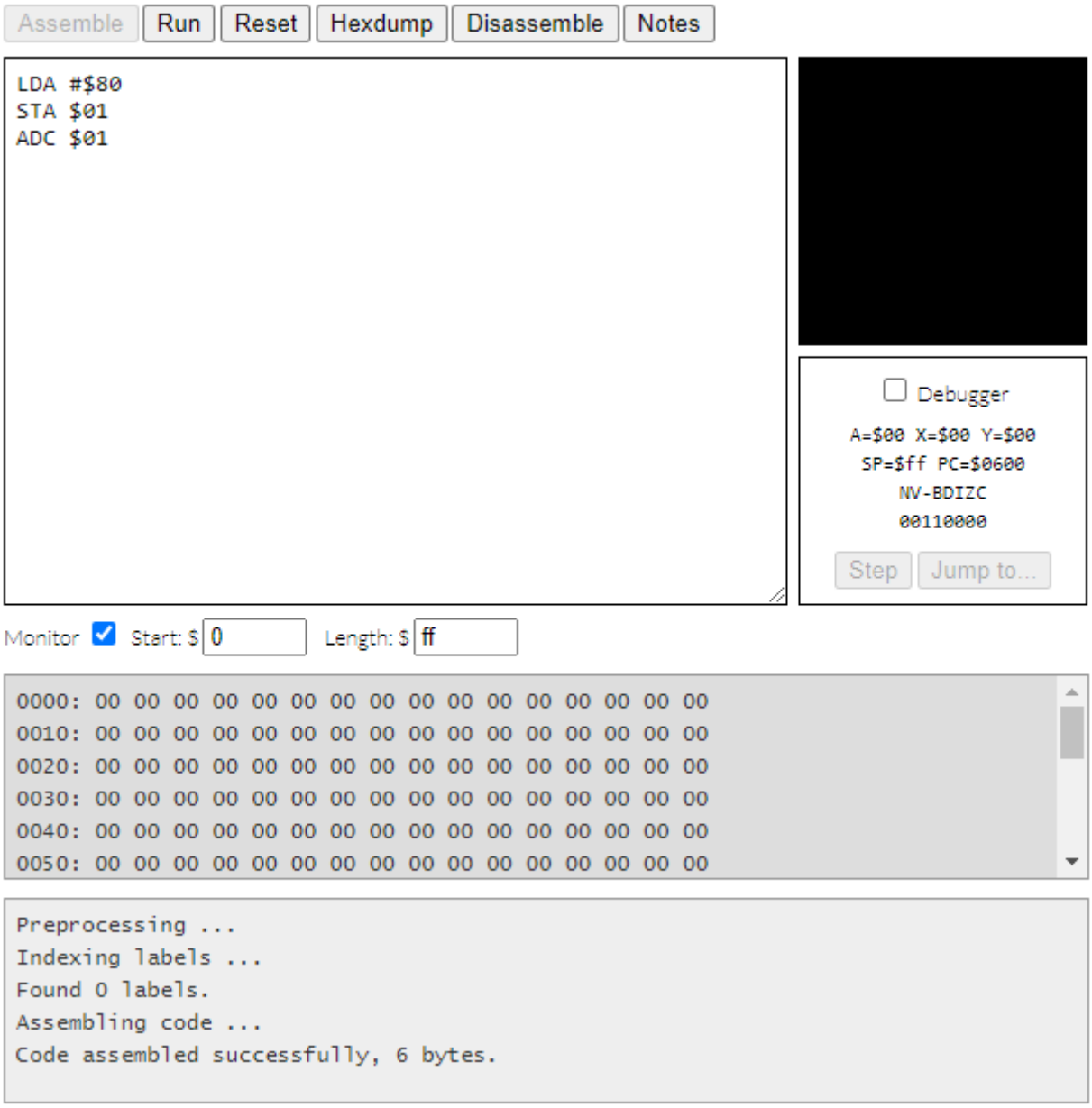
In the simulator below **type** the following code:

```
LDA #$80
STA $01
ADC $01
```

Enter the text manually as directed. Thanks! You'll be better off now for the effort!

The code as seen above has been entered. Next, click the **Monitor** ☐ checkbox and then the **Assemble** button. You should see the checkbox appearing as ☑ along with a set of numbers that represent values stored in various memory locations. The screen should now appear as seen on the next page. NOTE: Until the program is run, we do not know what values are in memory. It turns out, in this case at least, that website creator Nick Morgan initialized all values to **$00**. Whew! Thanks Nick.

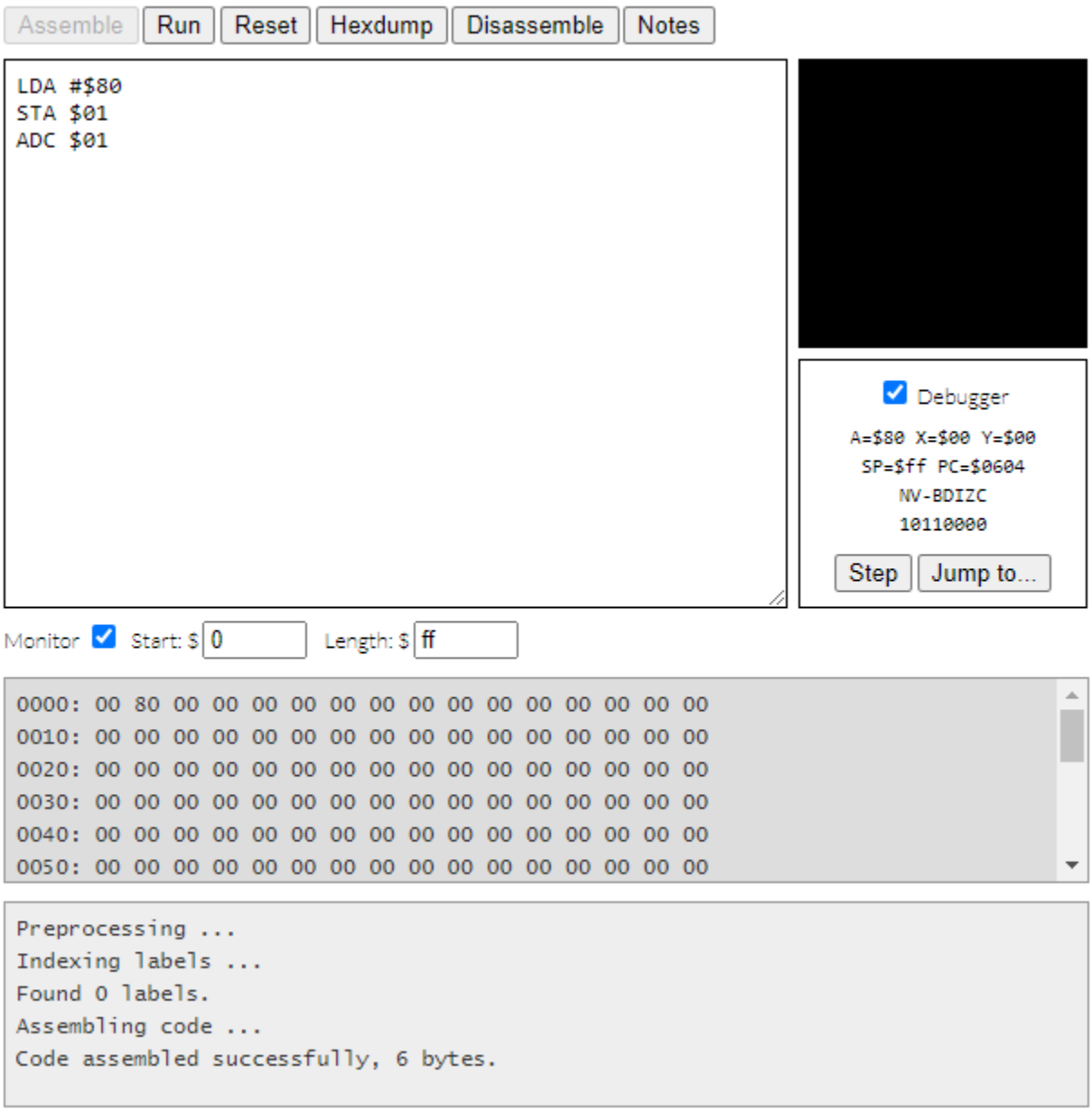
NOTE: A full listing of the 6502 Instruction Set is posted on “Canvas” as a “.PDF” format file for your perusal.



The screen after **Monitor** ☑ and **Assemble** have been activated.

Step 8 Observe the use of the symbol **#** in this new program compared to the program we have just stepped through. An important thing to notice here is the distinction between **ADC #$01** and **ADC $01**. The first, **ADC #$01**, adds the hexadecimal value **$01**to the contents of the **A** register, but the second adds the value stored at memory location **$01** to the **A** register.

Yikes, Mr. Bill! Do we even know what's stored there? Fortunately, the preceding instruction, **STA $01**, (**ST**ore **A**ccumulator value) when executed, stores the value currently in the Accumulator (in this case **$80**) in memory location **$01**.

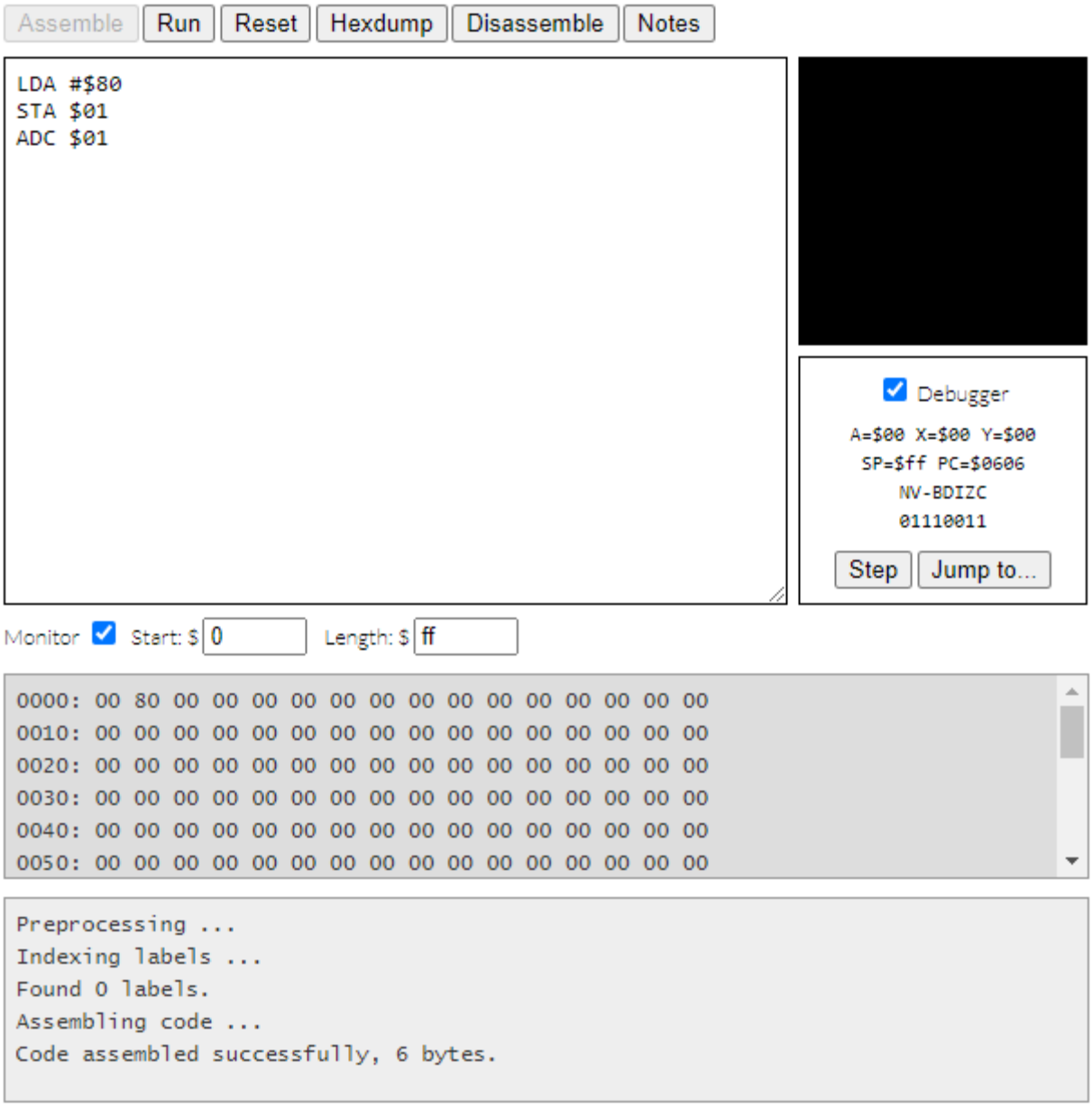


'**Step**' has been clicked twice and the instruction **STA $01** has been executed. Notice that a value of **#$80** (a numeric value, not a memory address) has been stored in memory location **$01**.

Step 9 In the screenshot above, **Assemble** has been clicked, the '**Debugger**' checkbox has been activated, and the User has stepped step through the first two instructions. The debugger monitor shows a section of memory and can be helpful to visualize the execution of programs. As you can see, the instruction **STA $01** stored the value of the **A** register at memory location **$01**.

Ask yourself: "What else happened (i. e. changed) when we executed **LDA #$80**?" & "Why?"

When we click **Step** again, the instruction **ADC $01** will add the value stored at the memory location **$01** to the **A** register. Note here that **$80** + **$80** should equal **$100**. However, because this value is larger than a byte (maximum possible unsigned value is 255 or **$ff**), the **A** register contains **$00** and the **Carry** flag is set. Notice as well that the **Zero Flag** is set. The **Zero Flag** is set by all instructions where the result in register **A** is zero. See the next page and observe the results.



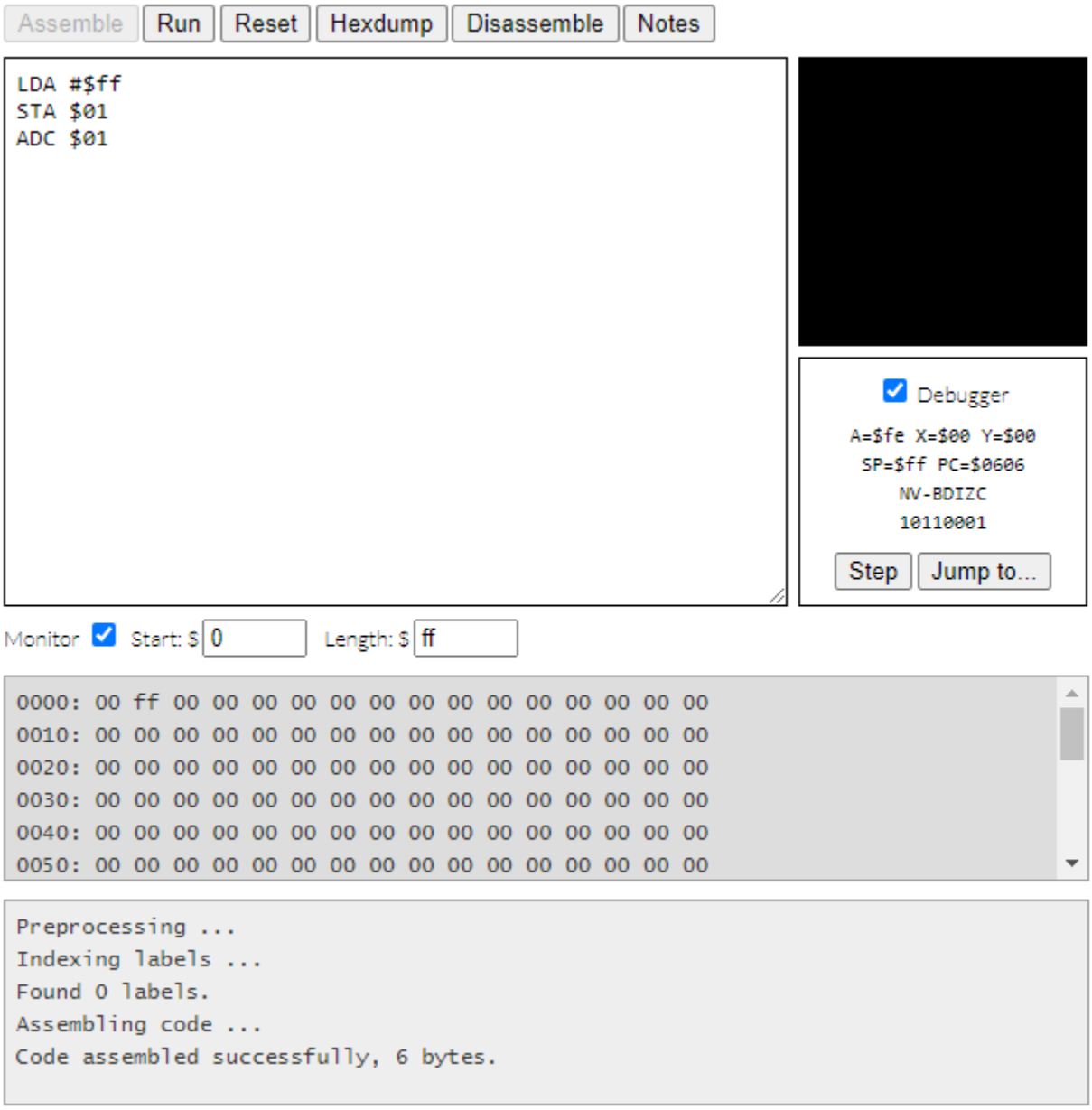
Instruction **ADC $01** has been executed and, for better or for worse, the results are in:

- Accumulator A has overflowed and now contains **$00**.
- Since the Accumulator's value is **$00**, the Zero Flag ('**Z**') is set.
- Since the Accumulator 'overflowed', the Carry Flag ('**C**') is set.
- We haven't discussed it yet, but there is a flag called the Overflow Flag ('**V**'). Notice that it too is set. We'll get to it when we discuss *signed* addition and subtraction.
- We haven't discussed it yet, but there is also a flag called the Negative Flag ('**N**') that is used to indicate that the value in the Accumulator is a negative number.
- Look at the next screenshot. Here, the value to be stored in the Accumulator has been changed to **$ff** (Binary **1111 1111**). In signed number notation, any value whose left-most digit is a logic **1** is considered to be a negative number. Notice here that **we** might consider **$ff** to be a positive number or maybe not. That decision is important in creating your programs and it is your responsibility to know and deal with the 'meaning' of each and every numeric value in your program. Notice that the **N** flag has been set.

Submit: Include in your submittal document a properly-labeled screenshot after you complete the steps above.

Another Example

Notice the new value (hex **$ff** or 255 decimal) that has been loaded into the Accumulator:

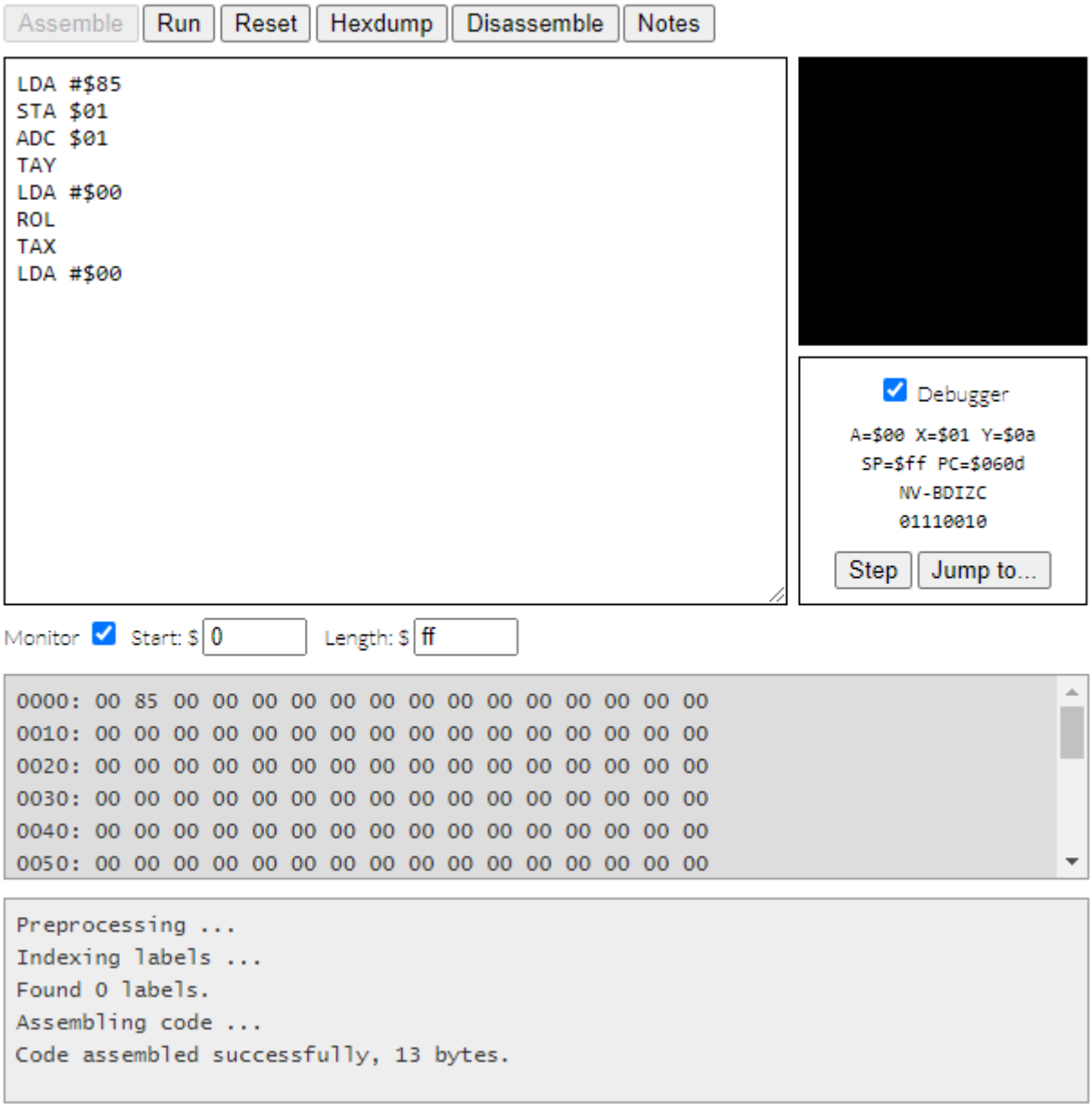


In this program, we have loaded **$ff** (decimal 255) into the Accumulator and then copied this value to memory location **$01**. We then add the Accumulator value **$ff** to **$ff** in memory location **$01** and get the sum **$fe** (decimal 254) in the Accumulator; the Carry Flag indicates an overflow. Notice the bits in the Flag Register:

The Carry Bit is high and the “Negative” bit (“**N**”) indicates that we have a negative number in the Accumulator. IMPORTANT: whether that is true of not (i. e. whether or not the Accumulator value is negative) depends upon how the programmer decided to specify numbers. Signed? Unsigned? That decision affects the way we treat Accumulator flags throughout the program!

Displaying Correct Sum

Enter the program sown below and then click Assemble, Monitor, and Debugger:



Before leaving this section (“Instructions”), let’s look at the simple program above, one that seeks to:

1. Add **$85** (decimal 133) to itself (sum will be **$10a**, decimal 266).
2. Store the low-order part of the sum (**$0a**, decimal 10) in register **Y**.
3. Store the Carry Bit in register **X** (decimal weighting is 256).
4. Clear the Accumulator so that the correct sum is displayed in the Debugger from left to right.
5. In the instructions seen above, note that the **LDA #$00** instruction loads a value of zero into the Accumulator without affecting the Carry Bit. Only the **N** and **Z** flags are affected.
6. The instruction **ROL** shifts the contents of the Accumulator to the left, bringing the Carry Bit in to become the least significant bit of the Accumulator. Since the Accumulator was just set to zero before this action is taken, the Accumulator contains either a value of **$01** (if the Carry was high from the previous operation) or **$00** (if the Carry was not set). This bit has a weighting of 256.
7. Notice that the Carry Bit is cleared by this operation.
8. The instruction **TAX** copies the Accumulator contents to the **X** register – for display purposes only.

Step 10 Refer back to the original code of Steps 1- 6. With your screen matching that seen in Step 2, delete the two dollar signs (**$**). Notice that the **Assemble** button is once again available for use. Click it and observe what happens.

Answer the following:

- Can you now run this modified program?
- What has happened?
- What do you conclude from your observation?
- What values are required at the two locations where the dollar signs were removed to make the program (a) assemble and (b) generate the same results as produced by the original program?

Step 11 Refer to the original code for the "Branching" section. Modify the original code as required to make the **Y** register act as a counter. Submit this new code as a text file. Name the file {Your Last Name} Lab 3 Step11.Txt. Be sure that all sections of your document that contain screenshots are labeled clearly.

Submit text files and your spiffy document to RSturlaCS130@GMail.com.

Original Code from 'Branching' Section

Optional

Step 12 The opposite of **ADC** is **SBC** (**S**u**B**tract with **C**arry). Write a program that uses this instruction.