Assignment 2:
Computer Architecture

Yazeed AlKhalaf

Course: CIS 304 - Computer Architecture
Instructor: Dr. Adeel Baig

Date: 21 Apr, 2024

Contents

1 Question 1:

A cache consists of 128 lines. The main memory contains 8192 blocks of 256 words each. Design the address format if the cache is

(a) Direct Mapped

(b) Associative Mapped

(c) Set Associative with four-line per sets

1.1 Shared Information:

- $cacheLines = 128$ cache line
- $blocks = 8192$ block
- $words = 256$ word
- One memory address bit length: $\log_2(blocks * words) = \log_2(8192 * 256) = \log_2(2097152) = 21$ bits

1.2 Part A: Direct Mapped

For the direct mapped cache, we need to know three things:

- Required cache lines bit length: $\log_2(cacheLines) = \log_2(128) = 7$ bits
- Required word offset bit length: $\log_2(words) = \log_2(256) = 8$ bits
- Required tag bit length is the remaining bits length: $21 - 7 - 8 = 6$ bits

The design is: **6 Tag Bits — 7 Cache Line Bits — 8 Word Offset Bits**

1.3 Part B: Associative Mapped

Associative mapped cache means the cache can store any block in any line. Therefore, we only need to know the word offset bit length:

- Required word offset bit length: $\log_2(256) = 8$ bits
- Required tag bit length is the remaining bits length: $21 - 8 = 13$ bits

The design is: **13 Tag Bits — 8 Word Offset Bits**

1.4 Part C: Set Associative with four-line per sets

Set associative means that the cache lines are split into sets, and each set has some lines, in our case 4 lines, and those are associative which means 4 blocks can exist at the same set.

And here since our lines are split into sets of four, we need to get the number of sets to be able to calculate the "cache set" bit length. To calculate the number of sets, we do the following: $(cacheLines \div 4)$. The 4 here is inferred from the "four-line per set" part.

Taking all of that in mind, we can get the following values:

- Number of cache sets: $128 \div 4 = 32$ cache sets
- Required "cache sets" bit length: $\log_2(32) = 5$ bits
- Required word offset bit length: $\log_2(256) = 8$ bits
- Required tag bit length is the remaining bits length: $21 - 5 - 8 = 8$ bits

The design is: **8 Tag Bits — 5 Cache Set Bits — 8 Word Offset Bits**

2 Question 2:

2.1 Introduction

THe goal of learning 6502 assembly language is purely academic, since we won't use it in our jobs, or perhaps our businesses. Since assembly is the lowest level language before the raw machine code, so it gives good understanding of the underlying things we we work with a fancy high-level language.

6502 assembly language was made with humans reading and writing it in mind. So learning it is easier and more fun than learning X86, as Nick Morgan claims. Also it is easier in general since the intrsuction set for 6502 is made up of 56 instructions. On the other hand, the x86 number of instructions can reach up to a 1000 instructions, and it is made for compilers to compile from high-level languages to it.

With that in mind, we can start the journey of exploring 6502 assembly language!

2.2 Our first program

The first program is a program that colors the first three pixels of the screen using obviously, 6502 assembly.

The code for this program is:

```
1 LDA #$01
2 STA $0200
3
4 LDA #$05
5 STA $0201
6
7 LDA #$08
8 STA $0202
```

The state of the simulator after assembling the code, but before running it is shown in **Figure 1** below. Notice the where the *PC* register's value, it is $0600.

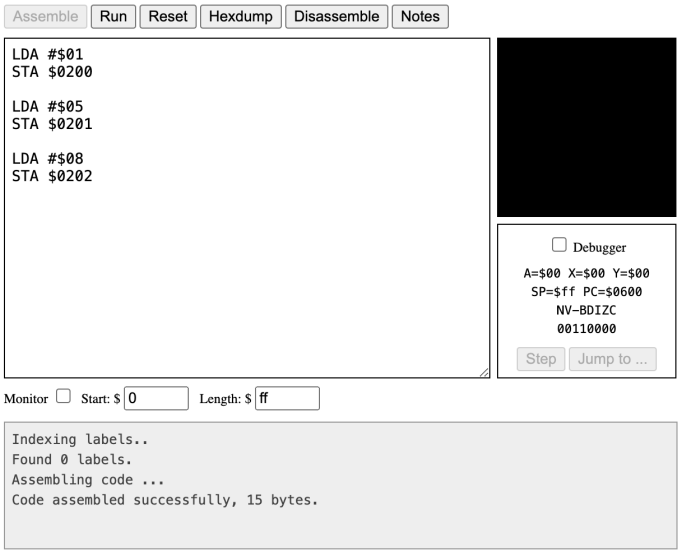

Figure 1: Initial state of the 6502 simulator with first program code

Now let us see what happened when I ran the code. The result is shown in **Figure 2** below. After the simulator is don executing we can see the changes that happened to the *PC* register. Of course other registers changed, but we will see them later when we try the debugging functionality.

Also the black screen on the top right corner of **Figure 2** shows three pixels colored, and that is what the first program instructed the 6502 simulator to do.

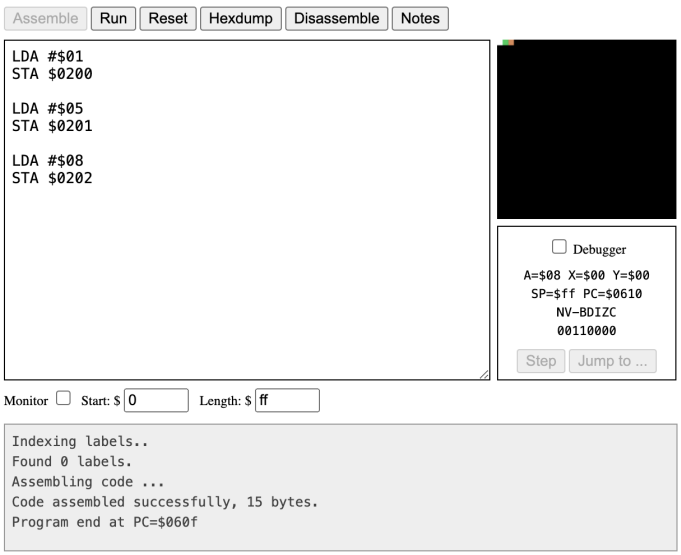

Figure 2: Result of running the first program in the 6502 simulator

Next we will be exploring the debugger functionality in this 6502 simulator. The debugger allows us to step through each CPU cycle manually to allow for debugging, hence the name "debugger". You can use use the debugger by checking the debugger option and then using the step button to step through the assembly code you wrote. You can notice the PC and any other registers you changed, and the memory places for the screen pixels change once you click the step button. I cliked it twice and see what happenned in **Figure 2** below. Notice the values of the A, the accumulator and PC, the program counter, registers.

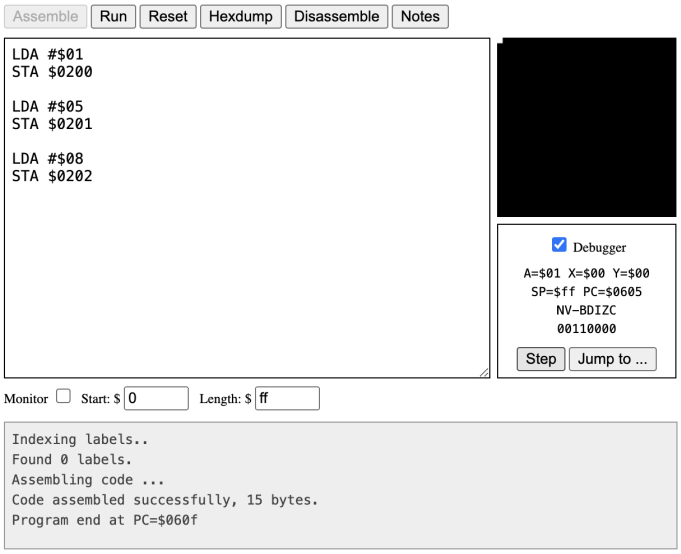

Figure 3: Result of running the first program with debugger in the 6502 simulator

After this initial trial with the simulator, I will be doing the following exercises and sharing the results below:

1. Try changing the colour of the three pixels.
2. Change one of the pixels to draw at the bottom-right corner (memory location $05ff).
3. Add more instructions to draw extra pixels.

2.2.1 Exercise 1

I will be editing the code to change the colors, so now the pixels will be showing the following colors in order from left to right: orange, light green, and light blue.

```
1 LDA #$08
2 STA $0200
3
```

```
4 LDA #$0d
5 STA $0201
6
7 LDA #$0e
8 STA $0202
```

After the running the code above, the result I got is shown in **Figure** 4.

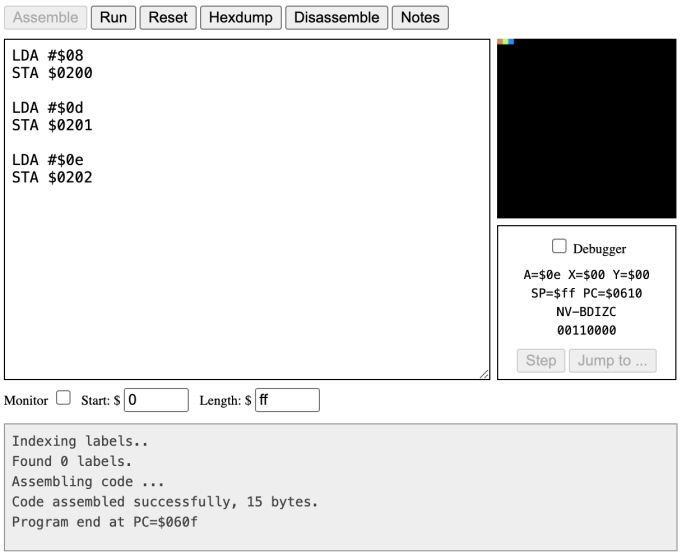

Figure 4: Result of running the first excercise in the 6502 simulator

2.2.2 Exercise 2

I will be editing the code from excercise 1 to make the green pixel that was in the middle of the three pixels we rendered before render at the bottom right of the screen. The memory address for that last pixel in the bottom right of the screen is $\$05ff$

```
1 LDA #$08
2 STA $0200
3
4 LDA #$0d
5 STA $05ff
6
7 LDA #$0e
8 STA $0202
```

After the running the code above, the result I got is shown in **Figure** 5.

Figure 5: Result of running the second excercise in the 6502 simulator

2.2.3 Exercise 3

After the first two exercises, I will be playing more with the pixels and changing the colors. I added pixels in the middle of the screen horizontally spaced equally. The color is both light red and white.

```
1  LDA #$08
2  STA $0200
3
4  LDA #$0d
5  STA $05ff
6
7  LDA #$0e
8  STA $0202
9
10 LDA #$0a
11 STA $0400
12 STA $0409
13
14 LDX #$01
15 STX $0410
16
17 STA $0417
18 STA $041f
```

The result is shown in **Figure 6**.

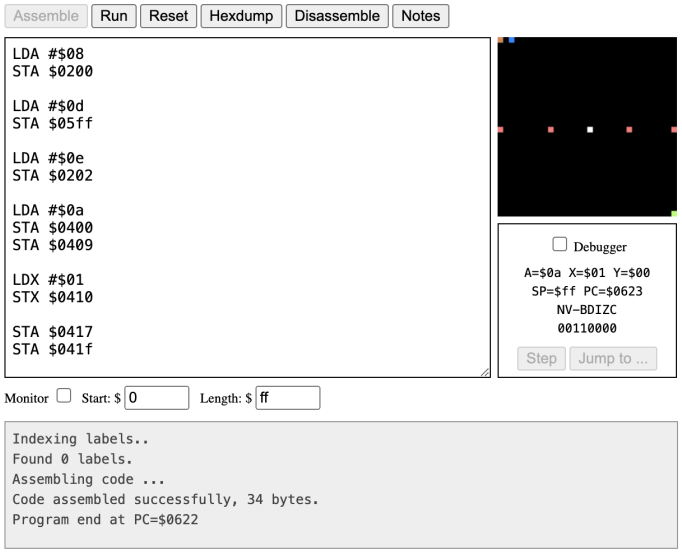

Figure 6: Result of running the third excercise in the 6502 simulator

2.3 Registers and flags

Here we learn about the registers and the flags of the CPU. Those can be seen in the simulator in the "processor status" section, shown in **Figure 7**. The registers this 6502 simulator has are:

- A register: the accumulator, holds a single byte.
- X register: a generic register, holds a single byte.
- Y register: another generic register, holds a single byte.
- SP register: stack pointer, basically this register is decremented every time a byte is pushed onto the stack, and decremented when a byte is popped off the stack.
- PC register: program counter, tells the CPU where the program is currently at in the memory, it is incremented every CPU cycle unless some JMP happens or something that interrupts the CPU.

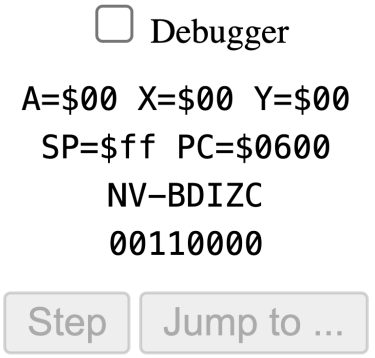

Figure 7: Processor status section in the 6502 simulator

Enough about the top section of the processor status section, let us talk about the bottom section that contains the processor flags. Each flag is one bit, so the seven flags we have can be represented as one byte. The flags are set byt the processor to give information about the previous instruction.

2.4 Instructions

Instructions in the 6502 assembly language are 56, we will be exploring some of them.

2.4.1 First Program

The first piece of code we have is the following:

```
1 LDA #$c0 ;Load the hex value $c0 into the A register
2 TAX ;Transfer the value in the A register to X
3 INX ;Increment the value in the X register
4 ADC #$c4 ;Add the hex value $c4 to the A register
5 BRK ;Break - we're done
```

Now let us use the debugger to step through this code above. **Figure** 8 shows the initial state of the simulator.

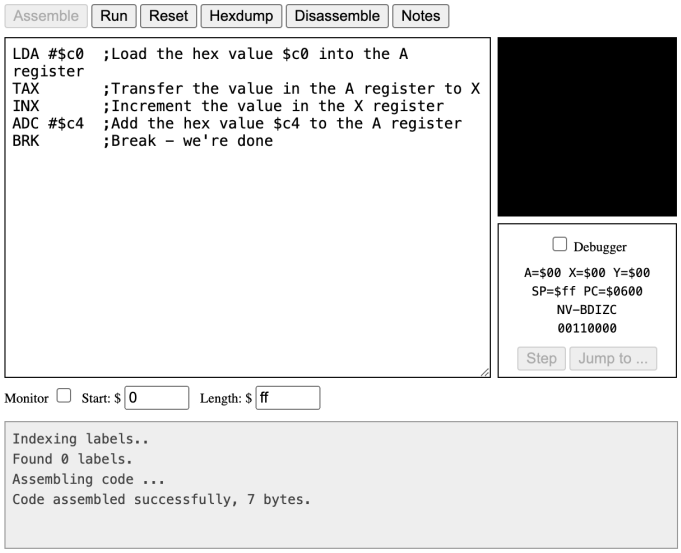

Figure 8: Initial state of first program in instructions in the 6502 simulator

We then step 4 times, let us view them below in order. The figures in order are: **Figure 9**, **Figure 10**, **Figure 11**, and **Figure 12**

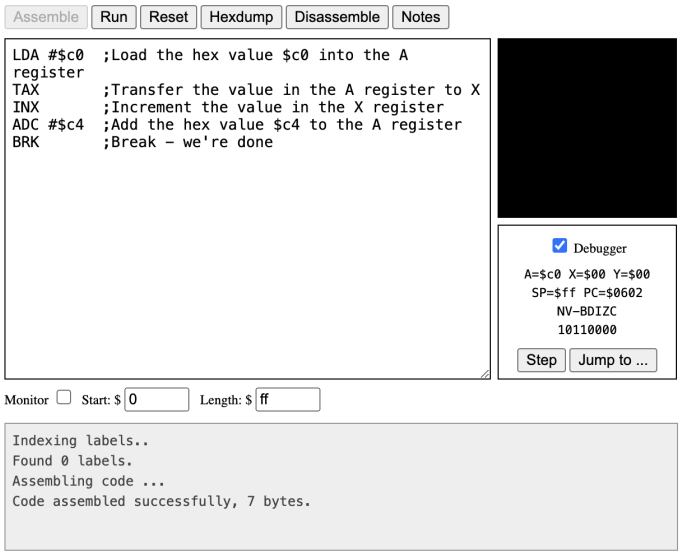

Figure 9: First step of first program in instructions in the 6502 simulator

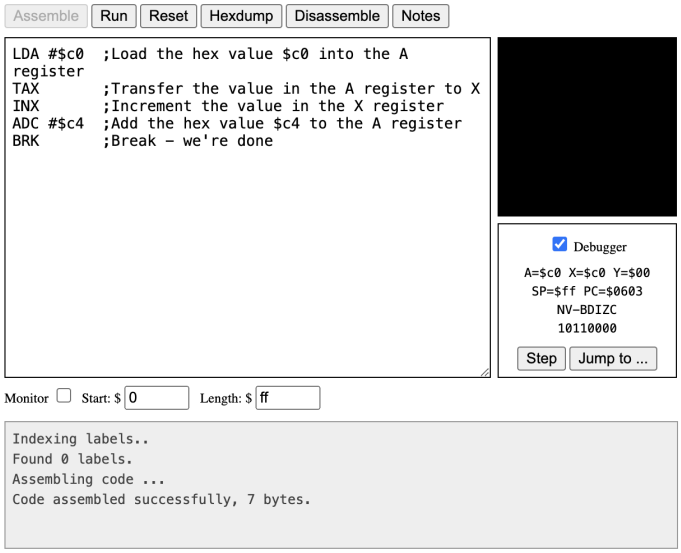

Figure 10: Second step of first program in instructions in the 6502 simulator

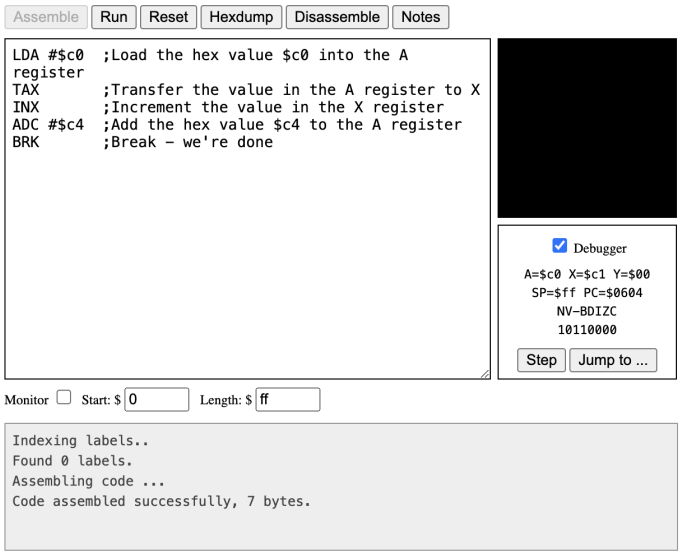

Figure 11: Third step of first program in instructions in the 6502 simulator

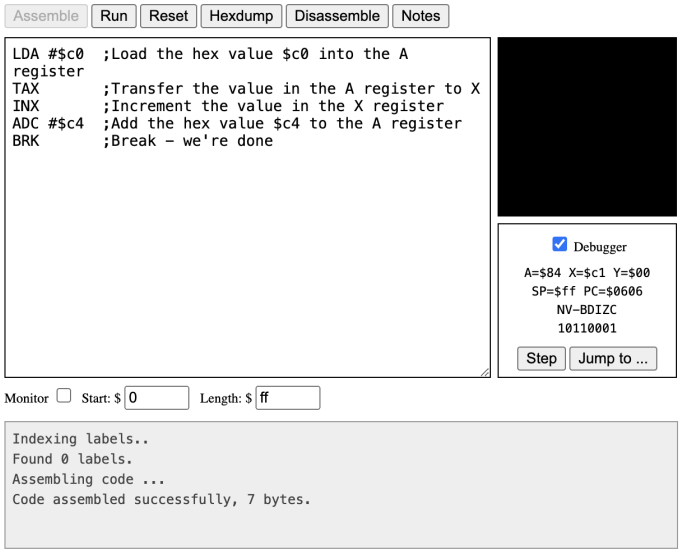

Figure 12: Fourth step of first program in instructions in the 6502 simulator

In the last step, **Figure 12**, you might expect that adding $\$c4$ to register A will yield the result \$184 but we are getting \$84. That is because the maximum number the register can handle is one byte, and \$184 is bigger than $\$FF$. And we can also notice that the processor carry flag was set to 1, and that's how we know that there is something weird. The carry flag is the last bit in the $NV-DBIZC$, hence the C.

2.4.2 Second Program

The second piece of code in this section is the following:

```
1 LDA #$80
2 STA $01
3 ADC $01
```

Notice an important thing regarding the ADC opcode, in the first example there was a # but now there is no #. When we have a leading #, the assembler will regards it as a number. But when it is directly stated like \$01, it replaces it with the value stored at that memory location.

Before running this code, we will tick the monitor checkbox so we can see the memory visually:

Figure 13: Initial state of second program in instructions in the 6502 simulator

Now we will be stepping through the code, three steps since they are three lines. Let us look at what happens to the memory in **Figure 15**.

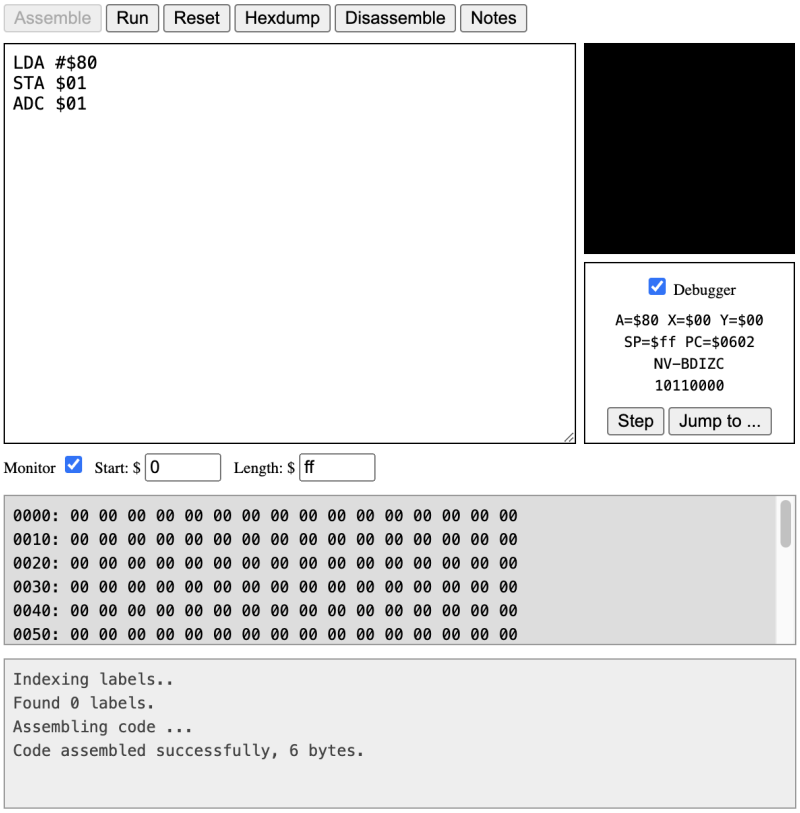

Figure 14: Step 1 of second program in instructions in the 6502 simulator

In **Figure** 14, the number 80 is loaded into register A. Notice how the memory is made up of ZEROs, that means it is empty for now.

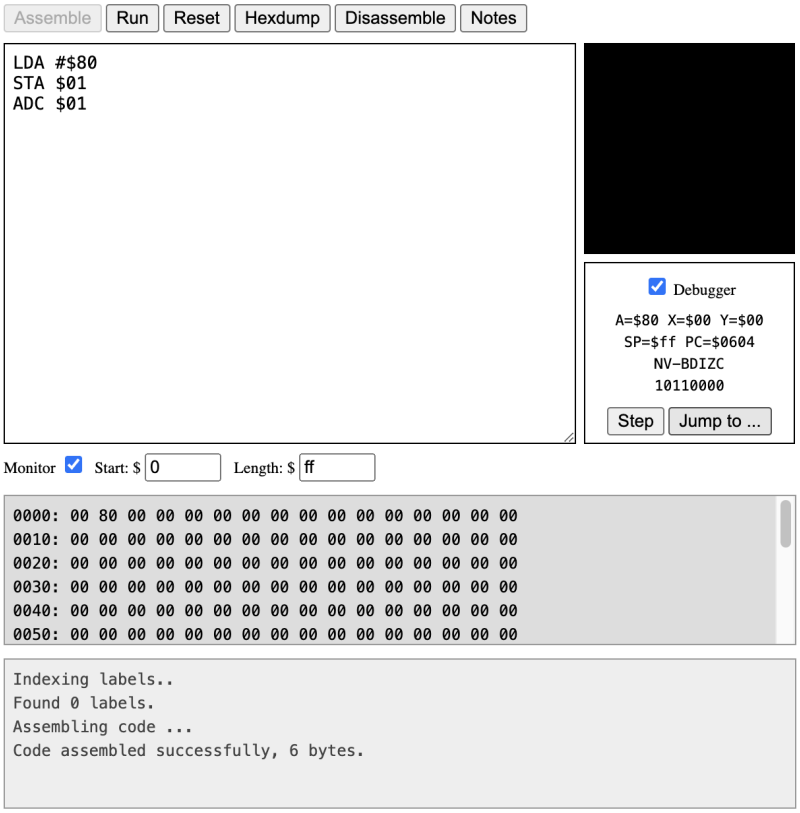

Figure 15: Step 2 of second program in instructions in the 6502 simulator

After that in **Figure** 15, the value held in register A, aka the accumulator, is saved at the memory location $01. You can notice it in the monitor in the first line.

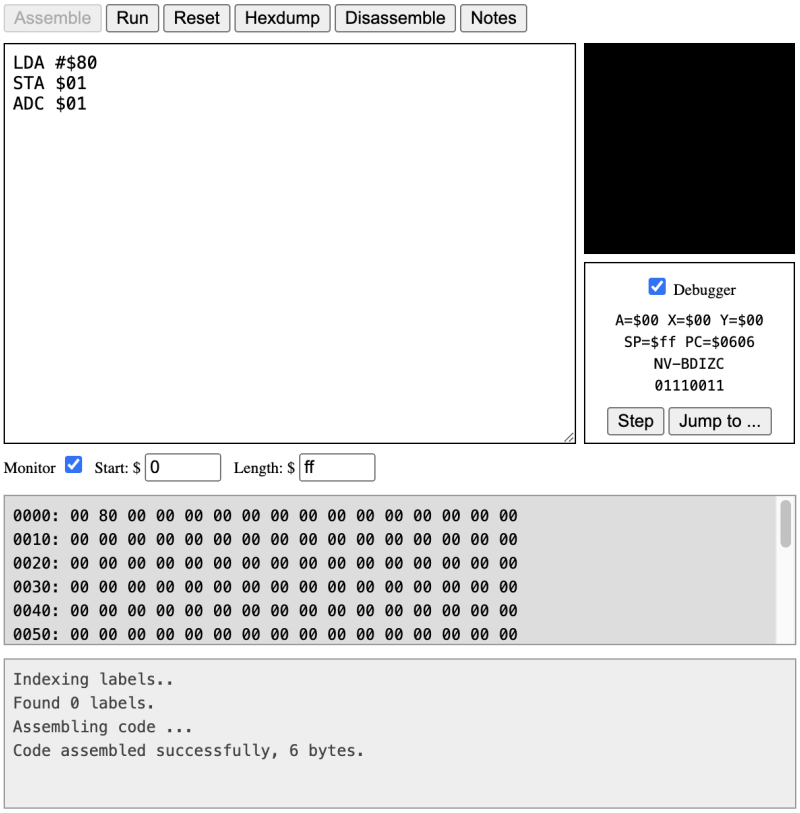

Figure 16: Step 3 of second program in instructions in the 6502 simulator

Finally, in **Figure 16**, the *ADC* opcode means "ADd with Carry". And this is adding the the value at $01 to the accumulator. Since $80 + $80 = $100, the accumulator should hold $100 but it doesn't because the value is bigger than a byte. Taking that into account, we can notice the things that changed in **Figure 16**. When this happens, the result is that the *A* register is set to $00 and the carry flag is set. The carry flag is the last flag in the flags byte. Another flag that has been set is the zero flag, since the result is zero.

After learning more about instructions, I will be doing the following exercises and sharing the results below:

1. You've seen TAX. You can probably guess what TAY, TXA and TYA do, but write some code to test your assumptions.

2. Rewrite the first example in this section to use the Y register instead of the X register.

3. The opposite of ADC is SBC (subtract with carry). Write a program that uses this instruction.

2.4.3 Excercise 1

I guess the meanings of the opecodes in the excercise before we start:

- TAY: means move the value of A to Y
- TXA: means move the value of X to A
- TYA: means move the value of Y to A

Here is a three pieces of code to test each one them, and I will be using the debugger to show its effect step by step.

```
1 LDA #$05
2 TAY
```

The result of running the code to test TAY is shown in **Figure** 17 and in **Figure** 18.

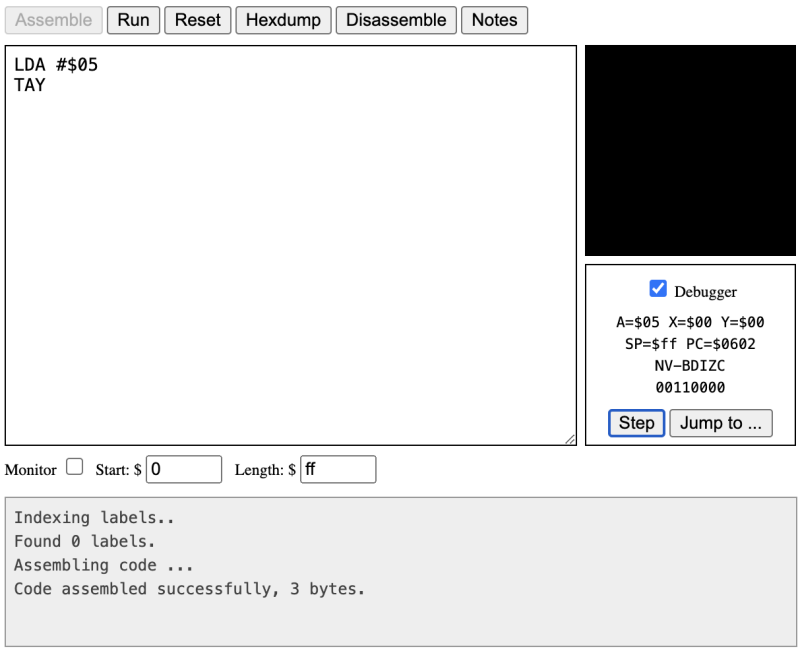

Figure 17: Step 1 of first excercise program related to TAY in instructions in the 6502 simulator

Figure 18: Step 2 of first excercise program related to TAY in instructions in the 6502 simulator

Notice how the value of register A is set in the first step in **Figure 17** and then is copied to the Y register which is shown in **Figure 18**.

The two next examples will be just the steps since they can be inferred easily based on the TAY explanation.

Figure 19 and **Figure 20** show the debugger exection of TXA, and the code is below:

```
1 LDA #$05
2 TAY
```

Figure 19: Step 1 of first excercise program related to TXA in instructions in the 6502 simulator

Figure 20: Step 2 of first excercise program related to TXA in instructions in the 6502 simulator

Figure 21 and **Figure 22** show the debugger exection of TYA, and the code is below:

```
1 LDA #$05
2 TAY
```

Figure 21: Step 1 of first excercise program related to TYA in instructions in the 6502 simulator

Figure 22: Step 2 of first excercise program related to TYA in instructions in the 6502 simulator

2.4.4 Excercise 2

2.4.5 Excercise 3