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.

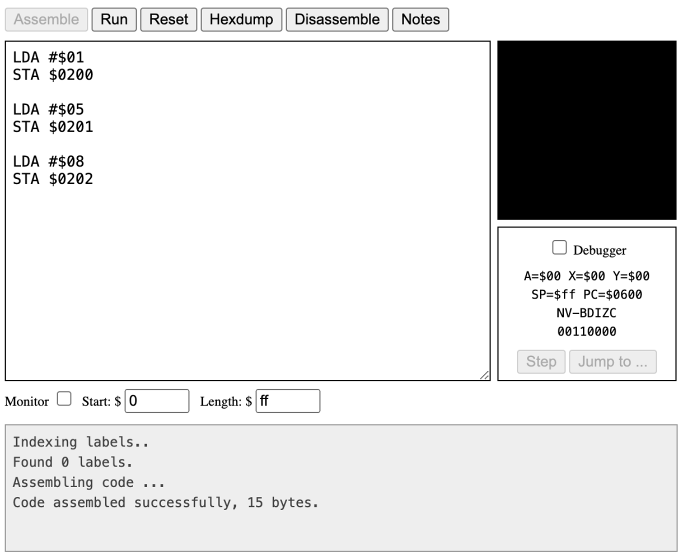
The code for this program is:

```
LDA #$01
STA $0200

LDA #$05
STA $0201

LDA #$08
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.



```
LDA #$01
STA $0200

LDA #$05
STA $0201

LDA #$08
STA $0202
```

```
A=$00 X=$00 Y=$00
SP=$ff PC=$0600
NV-BDIZC
00110000
```

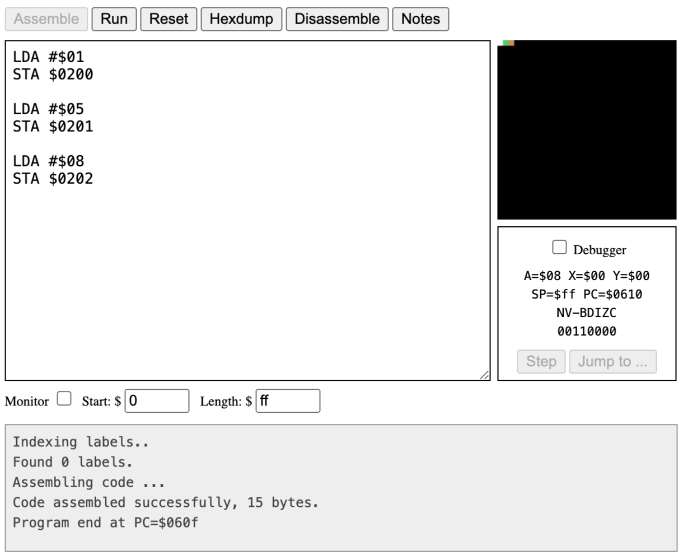
Start: $ 0 Length: $ ff

```
Indexing labels..
Found 0 labels.
Assembling code ...
Code assembled successfully, 15 bytes.
```

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.



```
LDA #$01
STA $0200

LDA #$05
STA $0201

LDA #$08
STA $0202
```

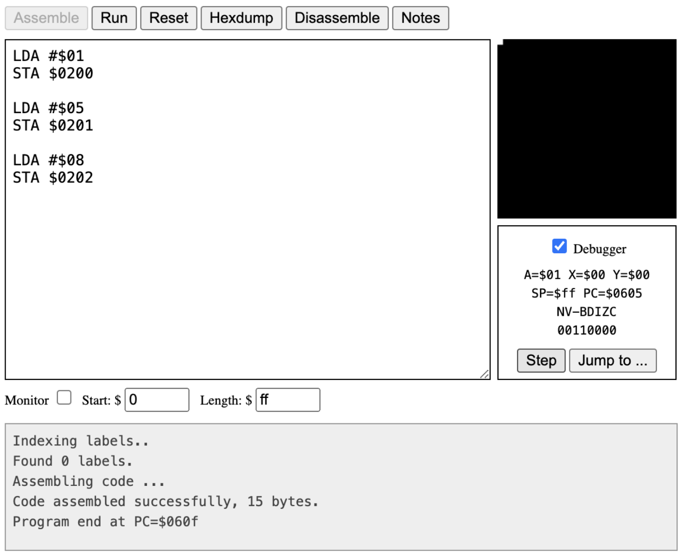
```
A=$08 X=$00 Y=$00
SP=$ff PC=$0610
NV-BDIZC
00110000
```

Start: $ 0 Length: $ ff

```
Indexing labels..
Found 0 labels.
Assembling code ...
Code assembled successfully, 15 bytes.
Program end at PC=$060f
```

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.



```
LDA #$01
STA $0200

LDA #$05
STA $0201

LDA #$08
STA $0202
```

```
A=$01 X=$00 Y=$00
SP=$ff PC=$0605
NV-BDIZC
00110000
```

Monitor ☐ Start: $ 0 Length: $ ff

```
Indexing labels..
Found 0 labels.
Assembling code ...
Code assembled successfully, 15 bytes.
Program end at PC=$060f
```

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.

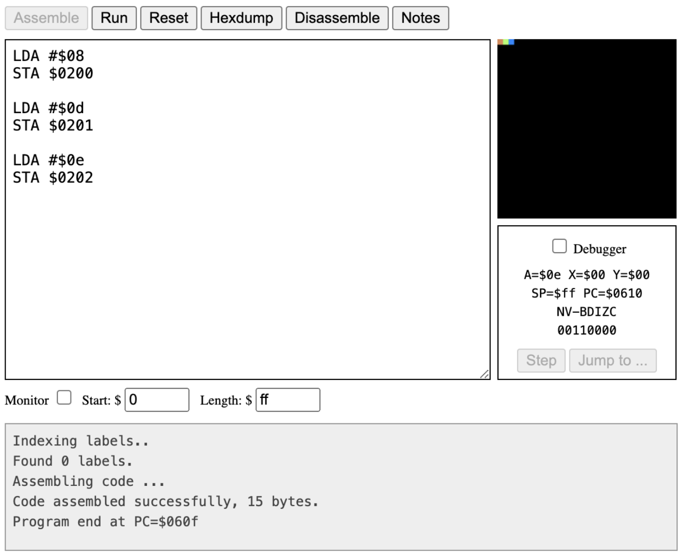
```
LDA #$08
STA $0200

```

```
LDA #$0d
STA $0201

LDA #$0e
STA $0202
```

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


Assemble Run Reset Hexdump Disassemble Notes

```
LDA #$08
STA $0200

LDA #$0d
STA $0201

LDA #$0e
STA $0202
```

Debugger
A=$0e X=$00 Y=$00
SP=$ff PC=$0610
NV-BDIZC
00110000

Step Jump to ...

Monitor Start: $ 0 Length: $ ff

```
Indexing labels..
Found 0 labels.
Assembling code ...
Code assembled successfully, 15 bytes.
Program end at PC=$060f
```

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

2.2.2 Exercise 2

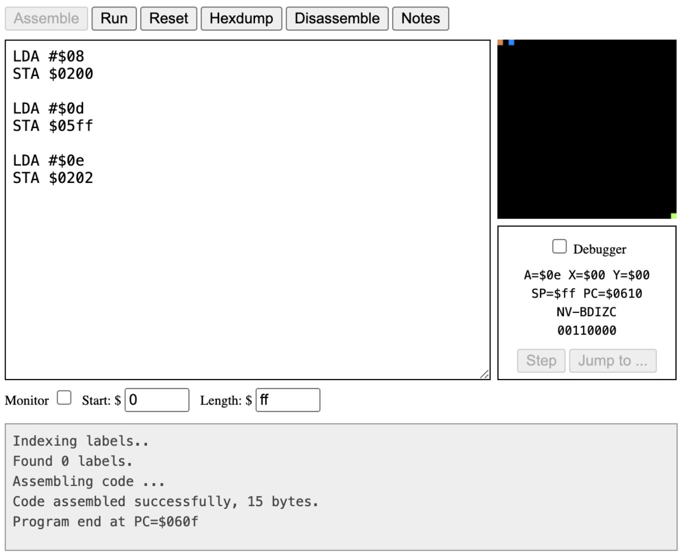
I will be editing the code from excercise 1 to make the green pixel that was in the middle of the three pixels we rendered before render at the bottom right of the screen. The memory address for that last pixel in the bottom right of the screen is $\$05ff$

```
LDA #$08
STA $0200

LDA #$0d
STA $05ff

LDA #$0e
STA $0202
```

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



```
LDA #$08
STA $0200

LDA #$0d
STA $05ff

LDA #$0e
STA $0202
```

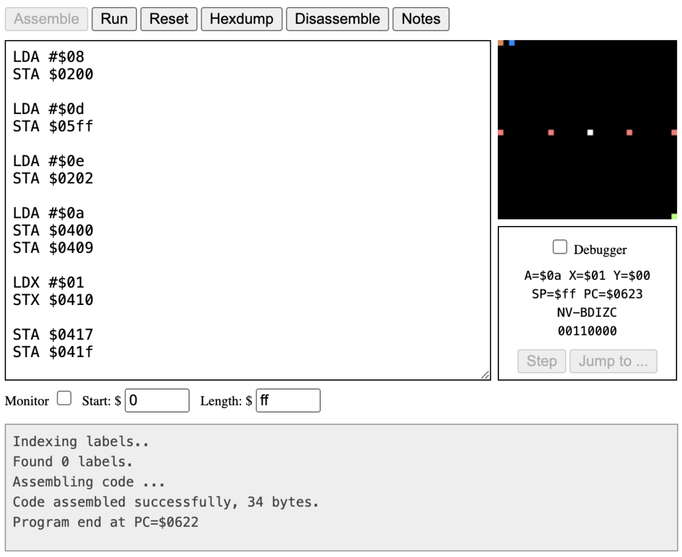
```
A=$0e X=$00 Y=$00
SP=$ff PC=$0610
NV-BDIZC
00110000
```

```
Indexing labels..
Found 0 labels.
Assembling code ...
Code assembled successfully, 15 bytes.
Program end at PC=$060f
```

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



```
LDA #$08
STA $0200

LDA #$0d
STA $05ff

LDA #$0e
STA $0202

LDA #$0a
STA $0400
STA $0409

LDX #$01
STX $0410

STA $0417
STA $041f
```

```
A=$0a X=$01 Y=$00
SP=$ff PC=$0623
NV-BDIZC
00110000
```

```
Indexing labels..
Found 0 labels.
Assembling code ...
Code assembled successfully, 34 bytes.
Program end at PC=$0622
```

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

2.3 Registers and flags

2.4 Instructions