

TI-83+ Z80 ASM for the Absolute Beginner

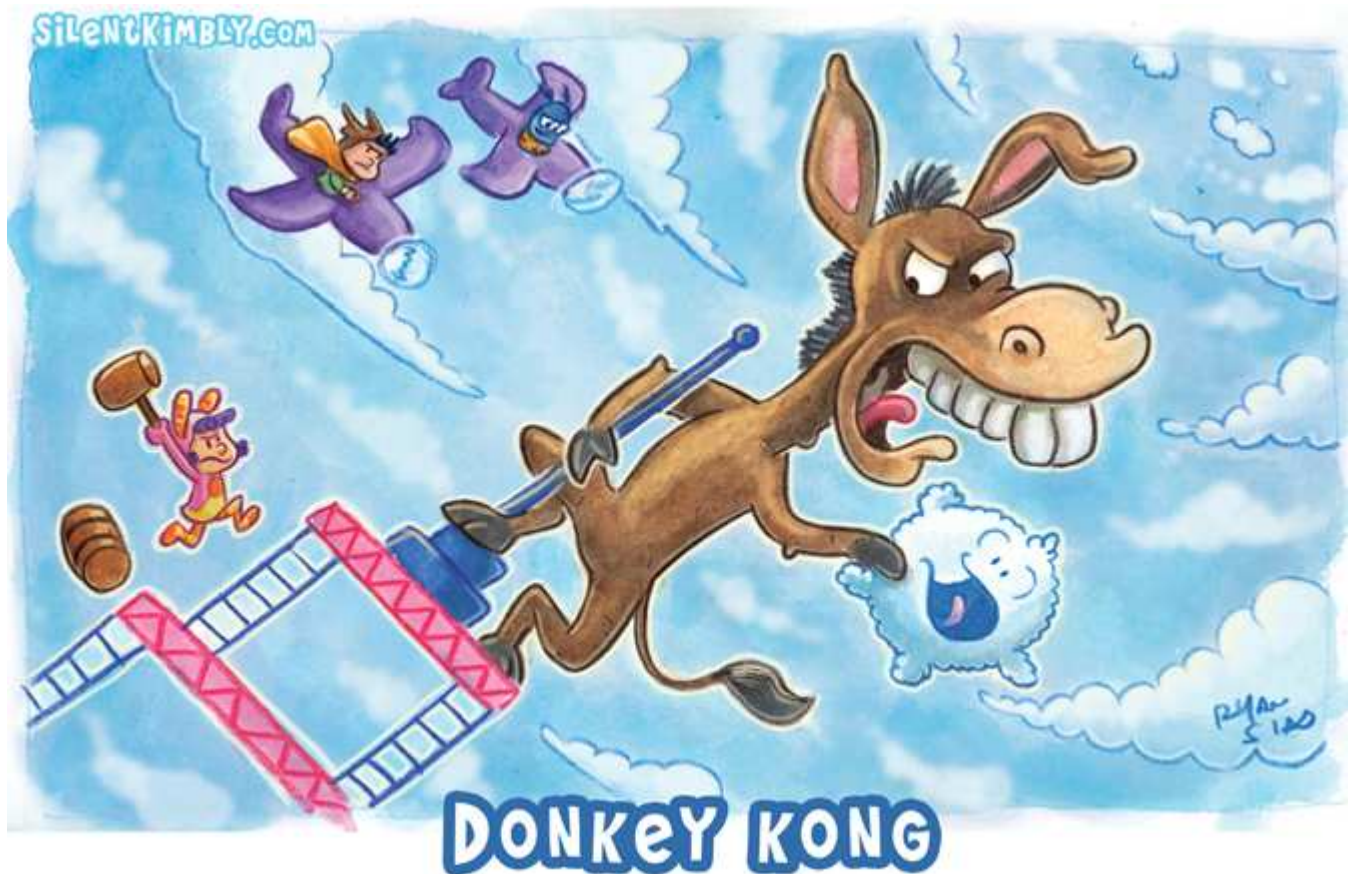
LESSON TWELVE:

- *Storing and Retrieving Values Larger Than One Byte*

STORING AND RETRIEVING VALUES LARGER THAN ONE BYTE

As a review from Lesson 5, you can store values to variables by using register A. You can also retrieve values from those variables by using register A.

Have you ever played Donkey Kong Country? (As a big thank you for the time I spent putting these lessons together, I'd love to see someone make Donkey Kong Country for a Ti-83+ or Ti-84+)



Donkey Kong can collect up to 100 bananas, and then he gains an extra life. Then the number of bananas is reset to zero. Thus, the

number of bananas can never exceed 255. So if someone decided to make a Donkey Kong Country game for a Z80 processor, register A would be perfect for this.

; Donkey Kong has collected 5 bananas

ld a, (Number_Of_Bananas)

add a, 5

ld (Number_Of_Bananas), a

For a game that requires a high score, however, one byte of numbers is not going to be enough...255 is way too much of a limit for high scores. And what about when your calculator has numbers such as 256, 341? That's WAY bigger than 255.

YET, the Z80 processor can do such things. In other words, since one-byte values aren't enough for everything, I'll show you throughout the lessons how to handle the bigger numbers. However, you need to be aware that the bigger the number is, the harder it is to work with said number. Even storing and retrieving values for two-byte numbers (0 – 65535) requires a little bit of work. This lesson, we will focus on variables holding 2-byte values.

You can only use Register A directly to store and retrieve values that are one byte in size. However, you can use HL, DE, and BC to store and retrieve values TWO bytes in size.

For this example program, I will be using hexadecimal numbers. (Remember that hexadecimal numbers are just numbers, a different “language” of numbers.) I am doing this because when you put two hexadecimal numbers together (such as 2 registers), you get the same number. If register H = \$EF and register L = \$23, HL = \$EF23. (Recall Lesson 12.) However, with decimal numbers, this is not the case. If H = 10 and L = 234, HL does not necessarily equal 10234. For this lesson, **it is important to visualize HL as H and L put together.**

```
#include "ti83plus.inc"
.org $9D93
.db  t2ByteTok, tAsmCmp

    ld de, $FC23          ; $FC23 in hexadecimal is equal to the number 64547
    ld (Score), de
    ld hl, (Score)

    B_CALL _DispHL
    B_CALL _getKey

    ret

Score:

.dw 0    ; Since we are storing and retrieving the value of a 2-Byte number, we use dw instead of db.
```

What happens is instead of using one register (A) to store a one-byte value, two bytes (H and L, D and E, etc) are used to store two one-byte values. Just remember that a two-byte variable requires—of course—two bytes of RAM.

So, are you ready to call me a liar? Seems too easy, huh? It does. Well, before you toss out the lesson and decide that I don't know my right hand from my left hand, try the next example program.

```
#include "ti83plus.inc"
.org $9D93
.db t2ByteTok, tAsmCmp

    ld hl, (Score)

    B_CALL _DispHL
    B_CALL _getKey

    ret

Score:

.db $FC
.db $23
```

What? When `_DispHL` is called, do you see the value 64547? No, you should see 9212.

I suspect that the Ti-83+ processor was designed on April Fool's Day, because the designers most likely decided to play a big joke on developers.

Here's the deal: LD HL, (Score) is the same as saying the following:

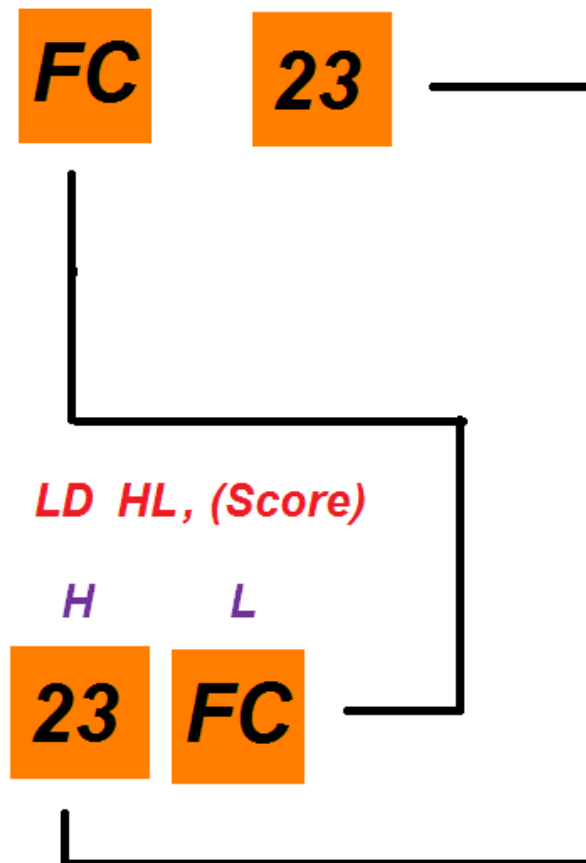
LD L, (Score)

LD H, (Score + 1)

**REMEMBER, THE HEXADECIMAL
NUMBER \$FC23 EQUALS THE
NUMBER 64547.**

**RAM
ADDRESS:**

SCORE SCORE + 1



Why is it not the other way around? Why not H first, and then L? That's the joke. The people who made the processor were trying to be funny. (Okay, so I don't know the real reason. But let me have some fun ☺)

This happens also with `ld DE, (Two-Byte Variable)` and `ld BC, (Two-Byte Variable)`. E is retrieved before D is, and C is retrieve before B is.

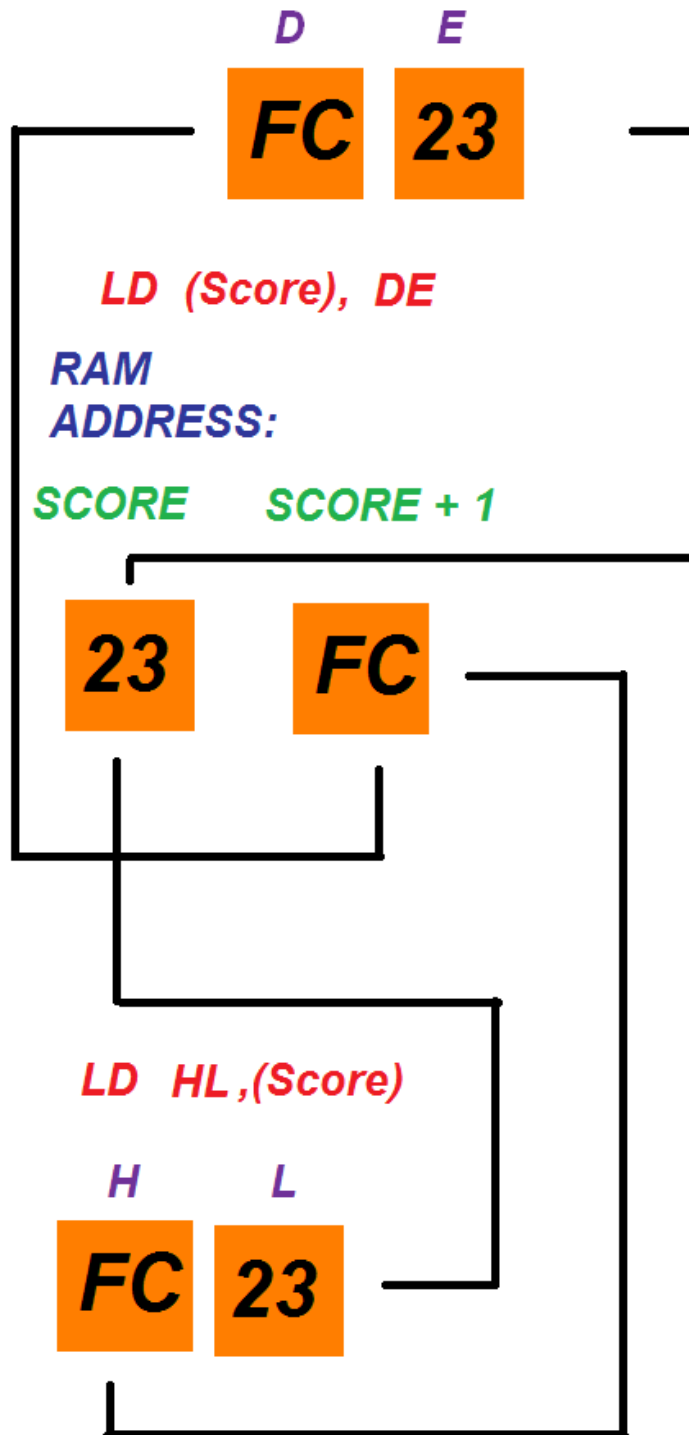
Why, then, did we not have any issues in the first example program? Because the registers are also switched with `ld (Two-Byte Variable), HL/DE/BC`. `ld (score), HL` is the same as saying

`ld (score), L`

`ld (score + 1), H`

As you will see in the diagram on the next page, (illustrating the first example program), this “reversing of values” occurs twice. So when you reverse the position of two values, and then reverse them again, you get back where you started.

**REMEMBER. THE HEXADECIMAL
REMEMBER, THE HEXADECIMAL
NUMBER \$FC23 EQUALS THE
NUMBER 64547.**



This “reversing of values” that occurs with 2-byte variables and registers is known as **little-endian**. I’m telling you this so that when you ask for help and people bring up this term, you know what they’re talking about. The important thing is, remember that if you store/retrieve values of 2-byte variables using two-byte registers, L will get stored/retrieved first, then H. E comes before D. C comes before B.

By the way, with SPASM, (be careful, this technique is for SPASM) you can replace the two .db statements with either .dw \$FE23 or .dw 64547. Why did I not do this in the first place? Because I wanted you to be aware of the little-endian problem. SPASM takes the .dw statement and places the numbers in the right positions so as to help avoid the little-endian problem. But NOT ALL COMPILERS DO THIS. Be very careful when working with .dw, and make sure you know how your compiler handles little-endians.