# *TI-83+ Z80 ASM for the Absolute Beginner*

## ***LESSON NINE:***

- *The Hexadecimal Numbering System*
- *263=1+7?*
- *Two-Byte Registers*
- *New Coding Instructions*

# THE HEXADECIMAL NUMBERING SYSTEM

So far, all the work we've done with numbers has been with the decimal numbering system. But we're getting close to the point where we need to look at binary numbers, meaning decimal numbers are difficult, or even impossible, for what we want to do.

However, a one-byte number requires eight 1s and 0s in binary, and a two-byte number requires 16 1s and 0s in binary. In addition, it's hard to instantly tell the equivalent decimal number by looking at a binary number. Needless to say, sometimes binary numbers can be inconvenient to work with.

Which is why we have another numbering system that solves these problems: hexadecimal. Binary has a maximum "ones", "tens", etc. digit of 1, and Decimal has a maximum ones, tens, etc. digit of 9. Hexadecimal has a maximum of 15.

But how do we represent this? We can't, for example, count 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, etc. because then by trying to have a maximum of 15 in the "ones" place, we end up with a tens place! Thus, to make sure we have 10, 11, 12, 13, 14, 15 in the ones place, we represent these numbers with letters. 10 = A, 11 = B, 12 = C, 13 = D, 14 = E, and 15 = F.

So, to count from 0 to 32 in hexadecimal, we have the following: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B, 1C, 1D, 1E, 1F, 20.

Okay, you're about to kill me for not explaining why hexadecimal is, in some situations, easier to work with than binary numbers. One digit of hexadecimal is equal to four digits of binary, which means you can store an 8-digit binary number in just two digits of hexadecimal. 1111 is equal to the hexadecimal digit F, so you can easily write 1111111111111111 as FFFF.

In addition, notice that ten of the sixteen possible values for a digit of hexadecimal—0 to 9—have a decimal equivalent! So it’s far easier to translate decimal to hexadecimal and hexadecimal to decimal quickly.

BUT, but but but but, you sadly have to discover the special, opportune moments for hexadecimal by yourself. Most of the time when you write a regular program, hexadecimal is used more for tradition (or for technical reasons I will not explain in these lessons) than for being a “reasonable numbering system” to use. For instance, hexadecimal is the numbering system traditionally used for numbering RAM addresses, and hexadecimal is the number system traditionally used to list the numerical equivalents of characters in a string. Yet these are only traditional, not convenient. As you program in ASM, you’ll learn when it’s appropriate to use hexadecimal.

Guess what? 40339 is 9D93 in hexadecimal!

## 263 = 1 + 7?

In lesson 4, I left you with the thought: if H = 0, and L = 6, HL = 6. But if H = 1 and L = 7, HL does not equal 17. So why is it that when you put H = 1 and L = 7 together that you don't get 17? You have to think of it in terms of binary numbers. H = 00000001 in binary, and 7 = 00001111 in binary. If you put them together, and come out with 0000000100001111, that number is indeed equal to 263.

This is also why H = 0 and L = 6 means HL = 6. H = 00000000 and L = 00001110, so HL = 0000000000001110.

Similarly, if we have a given value for HL, we can find out what H and L are equal to. Remember that 1 byte is equal to 8 bits, so by splitting the 16 bits of HL in half, we get H in the first half and L in the second. We can tell from the first eight bits of 0000000100001111 that H = 00000001 = 1 and L = 00001111, that last eight bits, which equals 7.

Let's look at the last paragraph in hexadecimal for practice. Remember that 1 byte is equal to 2 digits in hexadecimal, so by splitting the 4 digits of HL (hexadecimal) in half, we have H = 01 and L = 07 in hexadecimal. (Easier to visualize than binary, right?)

## TWO-BYTE REGISTERS

As a review from lesson 7, there are several 1-byte registers that can be combined into two-byte registers for important functions. H and L can be combined, D and E can be combined, and B and C can be combined. A and F can also be combined, but for very little purpose, which we will look at later. Avoid AF for right now.

You already know that HL is used to store RAM addresses. But it is also used for 2-byte math. If you need to solve a problem involving numbers too big to fit inside of register A, use HL to do your math.

DE is also used to store RAM addresses. It can be used to access RAM if need be, but you should avoid using DE for data storage and access until necessary. Rather, DE is kind of like HL's "partner." It is the register most often used to hold a RAM address when it needs to be saved for only a few instructions, and it is used to tell HL where to store data when large amounts need to be copied. It is also the register most often used when HL needs to do math. Why is DE so important in all of this? Because HL and DE *can exchange their values*. You can tell DE to hold whatever is inside of HL, and HL to hold whatever is inside of DE. No other registers, not even one-byte registers, allow you to do this so easily. So strictly speaking, don't use DE to hold and store data that HL can handle, but it should be the first register you use when HL needs a helping hand.

BC is another 2-byte register. It can do almost anything that DE can do, but it cannot exchange with HL. You should use BC in conjunction with HL only when DE is tied up. However, BC does have a special purpose, in that when you need to copy large amounts of data, BC is a **byte-counter**. It means just that: it holds how many bytes you need to copy, or how many bytes you need when a function you use has a parameter of a number of bytes.

There are other two-byte registers, we’ll look at those later.

## NEW CODING INSTRUCTIONS

The next couple few pages contain some instructions you can use with 2-Byte registers as parameters. 2-Byte value means a number up to 65535. HL BC DE means you can use either HL, DE or BC.

LD **HL BC DE, 2-Byte Value**

Stores any 2-Byte into HL, DE or BC.

Examples: LD HL, 256

LD DE, 9873

T-States: 10 Byte Storage: 3 Bytes

EX **DE, HL**

Exchanges the values of DE and HL. For example, if HL = 10 and DE = 257, EX DE, HL will mean DE = 10 and HL = 257. You must type in EX DE, HL; EX HL, DE will not work.

T-States: 4 Byte Storage: 1 Byte

ADD **HL, HL BC DE**

Adds HL, BC, or DE to HL. Notice you can even add HL to itself.

Examples:

```
LD HL, 1000
LD  DE, 1000
ADD HL, DE   ; HL now equals 2000
```

T-States: 11 Byte Storage: 1 Byte

INC **HL BC DE**

DEC **HL BC DE**

Increases HL, BC, DE by one, or decreases these by one.

Examples:

```
INC  HL
DEC  BC
```

T-States: 6 Byte Storage: 1 Byte