TI-83+ Z80 ASM for the Absolute Beginner

LESSON THIRTEEN:

- *Index Registers*
- *Floating-Point Numbers*

INDEX REGISTERS

Before I begin this lesson, I want to say that we are drawing to a close. Soon you will be ready to progress to the advanced world of ASM, which these lessons are meant to prepare you for. Wonderful job for making it this far!

Getting back on track, the Ti-83+ processor provides a couple of two-byte registers for special use. These registers are called index registers, and for a reason.

So, what do you think of when you hear the word "index?" Do you think of your index finger? Do you think about the index at the back of a school book? Or, how about indexing as in index cards?

Think about it. Let's say you are studying in a zoology class, and you need to prepare for a test that quizzes you on terms. You would probably write some terms on index cards. Then you keep these index

cards together as a group, and you can access your terms to prepare for the test simply by using your index cards.

Now, if you're a smart person, you'd probably keep them in alphabetical order. Like so:

Alligator, Bear, Chicken, Dragon, Elephant, Fish, Gorilla, Hippo, Iguana, Jackel…

So now, you have all these cards in alphabetical order. So what are you going to do if you only need to look up "Gorilla?" Are you going to flip each card in search of Gorilla? I certainly hope not! Because now that all the cards are in alphabetical order, you can find your card "Gorilla" almost instantly!

Let's apply this to programming. Suppose that you have some data that you need to keep organized. Here's a list of heights, in inches, for people in a classroom:

```
Heights:
.db 45, 45, 45, 45, 45, 46, 47, 47, 48, 48, 48, 48, 49, 49, 50, 50, 50, 50, 50, 50, 51, 51, 51, 52,
52, 53, 53, 54, 55, 58, 60
```

Let's say that a girl named Suzie is always the sixth person on the list whenever there is a list of people in the classroom. She is the sixth person in the attendance list, and she is the sixth person on the list of who's doing chores. This is because the teacher likes to keep her lists well-organized and easy to work with.

Furthermore, Bob is the 9th person in the list, Sarah is number 13, Chewy is number 18, and Jason is number 23.

So let's say that the teacher is using her calculator to find the height of these good little students because she forgot their heights. But she's using a Ti-83+ calculator with the Ti-Basic section corrupted, so she can only access this data by writing and running an ASM program. (Guys, let's just pretend, okay?)

What is she going to do? We did something like this once before when we looked at the characteristics of cars and trucks. Our solution was to LD HL, Label + Offset. For instance, Suzie is at Height + 5, and Chewy is at Height + 17.

However, at that time, we knew exactly where all the data was, because the location of the data was marked by a label. What if the location of this data of heights cannot be marked by a label? For example, let's say that the teacher's data is stored inside of an application variable, and this application variable is stored in RAM. As a programmer in Ti-Basic, I assume that you know that this application variable can be in different locations in RAM at any moment, depending on how much data/programs/etc. are added to RAM or deleted from RAM.

Thus, the location of the data is not consistent. Because of this, we can't say, for instance, LD HL, Height + 5 or LD HL, Height + 11. (Remember, Height, Height + 5 and Height + 11 are just representations of specific RAM locations, and just a reminder that this data in the application variable is not always in the same RAM location) We can still store the beginning of the data in HL, but we can't use labels to do so.

Let's say that the teacher found the location in HL. So now she could use the following code to access the heights of Suzie, Bob, Sarah, Chewy and Jason.

```
ld  de, 5
add  hl,  de          ; HL points to the first person in the list.
                      ; Since Suzie is number six in the list, we add 5 to HL
                      ; to get to the sixth person in the list.

ld  a, (hl)           ; Register A will contain the value of Suzie's height.

ld  de, 3
add  hl, de           ; Data location + 8
ld  b, (hl)

ld  de, 4
add  hl, de           ; Data location + 12
ld  c, (hl)

ld  de, 5
add  hl, de           ; Data location + 17
ld  d, (hl)

; BE CAREFUL!  Register D contains a value, so we need to save it!

push  de
ld  de, 5
add  hl, de           ; Data location + 22
pop de
ld  e, (hl)
```

Well, this works, but it's long and complicated. And there's another thing to consider: What if we need to access this data again very quickly? HL is used quite frequently in a program. Can you

imagine how much pushing, popping, and saving of values is required when we use HL to access heights very, very frequently?

Let's use IX to recover values from our data. This is where you'll see the use of IX and IY. Suppose that IX points to the beginning of wherever the teacher's data for heights is, just like HL did.

```
ld  a, ( IX + 5 )        ;Suzie's height
ld  b, ( IX + 8 )
ld  c,  ( IX + 12 )
ld  d,  ( IX + 17 )
ld  e,  ( IX + 22 )
```

Could it really be this simple? Yes, yes, yes! This is what is so special about IX and IY. You can use it to easily access data found in lists. You cannot do this with HL, at least not this way.

But there's more! (What? There's more?) IX and IY can do almost anything that HL can do. You can do math with IX and IY. You can point to RAM addresses using IX and IY. And, you can push and pop IX and IY. These registers are perfect to use if HL is tied up and you need to do some math.

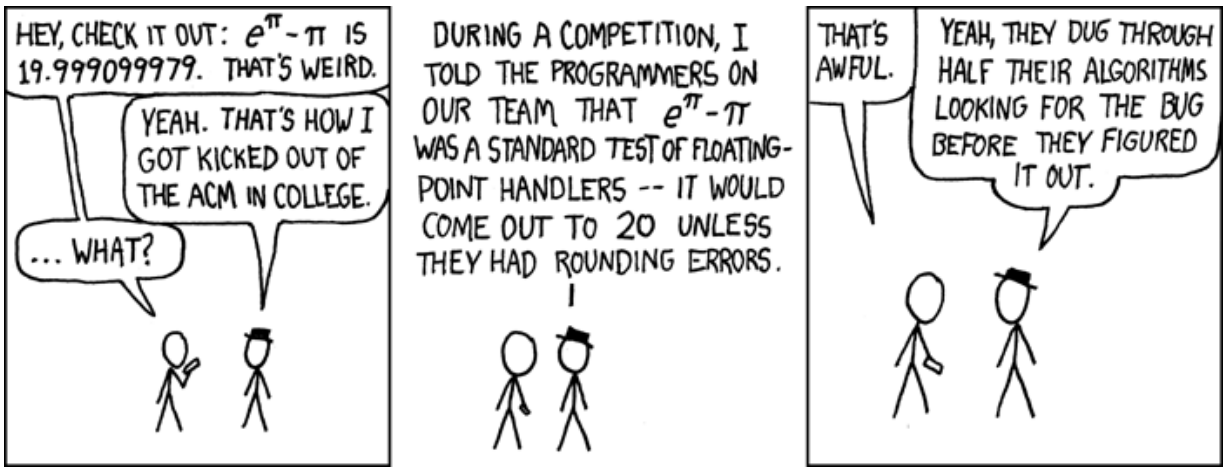
There must be a catch though, right? Right. In fact, there's several catches.

1. First and foremost, IX and IY are twice as slow as HL, and require more bytes for instructions than HL does. (For instance, ADD HL, DE requires only one byte, but ADD IX, DE requires two bytes.)
2. IX and IY cannot exchange their values with DE.

3. With HL, you are able to use H and L as individual bytes. You **can** do this with IX and IY, but then your program **will not** run on a Ti-Nspire. (See the Appendixes for more information.)
4. Use of IY is not recommended for beginners, and these lessons will not encourage changing its value. The Ti-83+ uses IY extensively, so if you try to mess with it without knowing how, you can crash your calculator. You will learn later what to do about IY, but don't try to show off by purposely and needlessly incorporating it into your program.
5. Finally, when you select an **offset** for IX/IY (such as IX + **16,** where 16 is the offset**),** your offset can only be a number from -127 to 128.

So, you should stick with HL when your processor needs beefy, intensive work done. IX (remember, be very, very cautious with IY) should be used for data access / math only when HL is tied up or when you need to access several areas in the same space of data. But just remember, if you frequently need to access an area with a whole bunch of data, IX will save you a lot of time and processing power, and keep you from going insane. As you program more and more, you will understand times when it is important to choose index registers over HL.

FLOATING-POINT NUMBERS



Since your Ti-83+ calculator is capable of doing decimal arithmetic, such as 1.5 + 2.25, it is logical to assume that you can use decimal numbers and floating-point numbers in Z80 ASM. For the purpose of these lessons, we are only going to work with Real Numbers, not Complex Numbers.

As you learned from previous lessons, you use Registers to do the actual computations for an ASM program. However, regular registers only store integers, and then these integers cannot be bigger than 65535 in value. Thus, we need to use special registers—located in the calculator's RAM—to work with floating-point numbers. These registers are called **OP Registers.**

The calculator provides 6 OP registers, called OP1, OP2, OP3, OP4, OP5, and OP6. These are really constants/labels for special areas of RAM, so you need to use them as such. For instance, you can say "LD HL, OP1," but you can't say "LD OP1, 4.352". So how do we store a value to an OP register? First we need to store our floating-point number in our program as a variable. Then we copy the number using LDIR, since LDIR is used to copy data from one RAM location to another RAM location.

Now, since a floating-point number is not an integer, we have a special way to tell the calculator exactly what the number is. A floating-point number on the calculator takes 9 bytes of RAM and can have 14 digits. As you probably know, the calculator only displays 10 digits, but you can use the last 4 digits in your number for extra precision.

The first byte will be one of four different values, but since we are working with only Real Numbers, we only need be concerned with two of them. This first byte will equal 0 if the number is positive, and 128 if the number is negative.

The second byte in your floating -point number is the number of digits and the exponent of your floating-point number. If this second byte is equal to 128, your floating point number has one digit, followed by a decimal point and then the rest of your 14 digits. If the second byte equals 129, you have two digits, a decimal point, and the rest of your digits. This pattern continues up to a value of 137, giving you a 10 digit floating-point number. As you probably noticed when performing calculations on a Ti-83+, this is the biggest number of digits you can display, after which you enter scientific notation starting at 10^{11}. So if your second byte equals 138 and up, you will have an exponent at the end of your number. (138 = 10^{11}, 139 = 10^{12}, 145 = 10^{18}, etc.) **If your second byte is less than 128,** your floating point number will have a negative exponent. For example, 127 will equal 10^{-1}.

The last 7 bytes of data are the actual digits of your floating point number. Be careful here. You store 2 digits per byte for 14 digits. How do you do that? By hexadecimal! Review the section on Hexadecimal if you need to, but recall that a digit of hexadecimal equals 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E or F. (It's not just 0,1 like binary or 0-9 as in decimal.) Now, did you notice that one byte of data has **2 digits** when you use hexadecimal? Make a note of that. So for every two digits, use

a hexadecimal number, with 0-9 for each digit. $24 will equal the 2-digit number 24, and $99 will equal the two-digit number 99. Pretty cool, huh?

So let's create some floating point numbers! Here are some examples.

```
Floating_Point_Number:
.db 0, 128, $92, $34, $57, $38, $12, $88, $21
; Equals 9.2345738128821

Floating_Point_Number_2:
.db 128, 120, $46, $20, $12, $00, $00, $00, $00
;Equals -4.62012 x 10^-8

Floating_Point_Number_3:
.db 0, 136, $12, $34, $48, $82, $99, $12, $12
;Equals 12344899,1212
```

Now, although the calculator provides many, many B_CALL functions to do math on floating-point numbers, we need our floating-point number stored in OP1 and, in some cases, OP2. We use LDIR for this.

Rather than go into details, I'll end this lesson with an example program with some B_CALL functions and lots of comments. We have a floating-point number, 1.5234523452345. We will start by displaying the floating-point number. Then we will multiply it by 3, and display the result. Add 1, display the result. Subtract 2, display the result. Finally, divide by 4.23 and display the result.

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

        ; Our floating point number, 1.5, is stored as a variable two pages below

        B_CALL _ClrLCDFull
        ld HL, Starting_Number
        ld DE, OP1
        ld BC, 9                ;A floating-point number is nine bytes long
        ldir

        SET fracDrawLFont, (IY + fontFlags) ;This allows us to display our floating-point number
                                            ;as big text on the home screen.  Otherwise,
                                            ;the number will display as small text.
        ld hl, 0                        ;Where we want to display the result on the screen
        ld (penCol), hl

        ld a, 11                ;Don't worry about this for now.  A should always be this value before
                                ;DispOP1A for the purpose of these lessons, in order to display 10 digits
        B_CALL _DispOP1A
        B_CALL _getKey
```

```
LD HL, Multiply_OP1_By_Three
LD DE, OP2          ;Multiplication requires two numbers, so OP2 holds our second one.
LD BC, 9
LDIR
B_CALL _FPMult      ;Multiply 1.5 by 3
ld a, 11            ;Displays 10 digits
ld hl, 0                    ;Where we want to display the result on the screen
ld (penCol), hl
B_CALL _DispOP1A
B_CALL _getKey

LD HL, Add1
LD DE, OP2          ;Addition two numbers, so OP2 holds our second one.
LD BC, 9
LDIR
B_CALL _FPAdd
ld a, 11            ;Displays 10 digits
ld hl, 0                    ;Where we want to display the result on the screen
ld (penCol), hl
B_CALL _DispOP1A
B_CALL _getKey

LD HL, Subtract2
LD DE, OP2          ;Subtraction requires two numbers, so OP2 holds our second one.
LD BC, 9
LDIR
B_CALL _FPSub
ld a, 11            ;Displays 10 digits
ld hl, 0                    ;Where we want to display the result on the screen
ld (penCol), hl
B_CALL _DispOP1A
B_CALL _getKey

LD HL, Divide_By_4_23
LD DE, OP2          ;Division requires two numbers, so OP2 holds our second one.
LD BC, 9
LDIR
B_CALL _FPDiv       ;Multiply 1.5 by 3
ld hl, 0                    ;Where we want to display the result on the screen
ld (penCol), hl
ld a, 11            ;Displays 10 digits
B_CALL _DispOP1A
B_CALL _getKey

ret
```

```
Starting_Number:

        .db 0, 137, $15, $23, $45, $23, $45, $23, $45

Multiply_OP1_By_Three:

        .db 0, 128, $30, $00, $00, $00, $00, $00, $00

Add1:

        .db 0, 128, $10, $00, $00, $00, $00, $00, $00

Subtract2:

        .db 0, 128, $20, $00, $00, $00, $00, $00, $00

Divide_By_4_23:

        .db 0, 128, $42, $30, $00, $00, $00, $00, $00
```